

University of Denver

**Digital Commons @ DU**

---

Electronic Theses and Dissertations

Graduate Studies

---

1-1-2019

## Improvements of and Extensions to FSMWeb: Testing Mobile Apps

Ahmed Fawzi Al haddad  
*University of Denver*

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



Part of the [Digital Communications and Networking Commons](#)

---

### Recommended Citation

Al haddad, Ahmed Fawzi, "Improvements of and Extensions to FSMWeb: Testing Mobile Apps" (2019).  
*Electronic Theses and Dissertations*. 1639.  
<https://digitalcommons.du.edu/etd/1639>

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](mailto:jennifer.cox@du.edu), [dig-commons@du.edu](mailto:dig-commons@du.edu).

# Improvements of and Extensions to FSMWeb: Testing 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

Ahmed Fawzi ALhaddad

August 2019

Advisor: Prof. Anneliese Andrews

© Copyright by Ahmed Fawzi ALhaddad, 2019

All Rights Reserved

Author: Ahmed Fawzi ALhaddad  
Title: Improvements of and Extensions to FSMWeb: Testing Mobile Apps  
Advisor: Prof. Anneliese Andrews  
Degree Date: August 2019

## Abstract

A mobile application is a software program that runs on mobile device. In 2017, 178.1 billion mobile apps downloaded and the number is expected to grow to 258.2 billion app downloads in 2022 [19]. The number of app downloads poses a challenge for mobile application testers to find the right approach to test apps. This dissertation extends the FSMWeb approach for testing web applications [50] to test mobile applications (FSMApp). During the process of analyzing FSMWeb how it could be extended to test Mobile Apps, a number of shortcomings were detected which we improved upon. We discuss these first. We present an approach to generate black-box tests to test fail-safe behavior for web applications. We apply the approach to a large commercial web application. The approach uses a functional (behavioral) model to generate tests. It then determines at which states in the execution of behavioral test failures can occur and what mitigation requirements need to be tested. Mitigation requirements are used to build mitigation models for each failure type. From those mitigation models failure mitigation tests are generated.

Next, this dissertation provides an approach for selective black-box model-based fail-safe regression testing for web applications. It classifies existing tests and test requirements as reusable, retestable, and obsolete. Removing reusable test requirements reduces test requirements between 49% to 65% in the case study. The approach also uses partial regeneration for new tests wherever possible. Third, we present the new FSMApp approach to test mobile applications and compare the



approach with several other approaches [88, 37]. A number of case studies explore applicability, scalability, effectiveness, and efficiency of FSMApp with other approaches. Future work makes suggestion on how to improve test generation and execution efficiency with FSMApp.

## Acknowledgements

I would like to express my deepest appreciation to my advisor Professor Anneliese Andrews for her tremendous support during my Ph.D. adventure. I cannot thank you enough for your patience, encouragement, leadership, and extensive knowledge. Your continuous assistance and guidance aided me to expand and challenge my research dissertation. I am grateful for having you as my advisor for my Ph.D. research.

In addition, I would like to thank my committee members, Professor Gareth Eaton, Professor Scott Leutenegger and Associate Professor Christopher GauthierDickey, for believing in me by being members in my oral defense committee. I truly appreciate your time and assistance. My special thanks to my parents, siblings, grandparents, friends, and colleagues for their fantastic support during my academic journey.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem	1
1.2	Existing Body of Work Related to FSMWeb	3
1.3	Research Agenda	4
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Black-Box Model-Based Testing	7
2.2	Testing Mobile Apps	9
2.2.1	MBT and App Testing	9
2.2.2	General App Testing Tools	26
2.3	Regression Testing (RT)	30
2.4	Fail-Safe Testing	32
<b>3</b>	<b>Original Approach</b>	<b>36</b>
3.1	Testing Fail-Safe Behavior w/FSM Web	36
3.1.1	Process	36
3.2	FSMWeb Approach	37
3.2.1	Example	41
3.3	External Failures (F), State-Event matrix (SE)	42
3.4	Potential Failure Scenarios	44
3.5	Generate Test Requirements	45
3.6	Mitigation Requirements and Models	45
3.7	Weaving Rules and Test Generation	48
3.8	Generate Tests, Execute and Validate	49
<b>4</b>	<b>Approach Improvements</b>	<b>50</b>
4.1	Mortgage System Case Study: Fail-Safe Testing	50
4.1.1	Case Study Research Questions	50
4.1.2	Mortgage System	51
4.1.3	FSMWeb Behavioral Model	54
4.1.4	Failure Applicability and Mitigation Requirements	62
4.1.5	Test Requirements and their Effectiveness	64
4.1.6	Comparison of Effort: GA vs. Exhaustive Search	71

4.1.7	Findings . . . . .	72
4.1.8	Threats to Validity . . . . .	73
4.2	Regression Testing Approach . . . . .	75
4.2.1	Process . . . . .	76
4.2.2	Detailed Approach . . . . .	80
4.2.3	Discussion . . . . .	100
4.2.4	Comparison to Earlier Work . . . . .	103
4.3	Mortgage System Case Study: Regression Test . . . . .	106
4.3.1	Case Study Objectives . . . . .	106
4.3.2	Case Study Research Questions . . . . .	107
4.3.3	General Description . . . . .	107
4.3.4	RQ1: Applicability . . . . .	108
4.3.5	RQ2: Efficiency of Selective Regression Testing . . . . .	112
4.3.6	RQ3: Genetic Algorithm vs. Coverage Criteria . . . . .	113
4.3.7	RQ4: Reusable Failure Scenarios . . . . .	113
4.3.8	Threats to Validity . . . . .	114
4.3.9	Practical Considerations . . . . .	115
<b>5</b>	<b>Extensions: Testing Mobile Apps . . . . .</b>	<b>118</b>
5.1	Testing Process for Mobile Apps . . . . .	118
5.2	Example Used to Illustrate Approach . . . . .	120
5.3	Phase 1: Build Model . . . . .	120
5.3.1	Partition the Mobile App into Clusters ( <i>Cs</i> ). . . . .	120
5.3.2	Define Logical App Pages ( <i>LAPs</i> ) and Input-Action Constraints for Each . . . . .	128
5.4	Phase 2: Generate Test Sequences . . . . .	146
5.4.1	Paths through FSMs/AFSM . . . . .	146
5.4.2	Path Aggregation . . . . .	149
5.4.3	Input Selection . . . . .	156
5.5	Phase 3: Execute and Validate Tests . . . . .	157
<b>6</b>	<b>Comparing FSMApp and other Approaches . . . . .</b>	<b>163</b>
6.1	Event Sequence Graph (ESG) Method . . . . .	163
6.2	A GUI Crawling-Based Technique for Android Mobile Application Testing . . . . .	175
6.3	Comparison of Results . . . . .	183
<b>7</b>	<b>FSMApp Validation via Case Studies . . . . .</b>	<b>189</b>
7.1	Rationale . . . . .	189
7.2	Case Study Objectives . . . . .	189
7.3	Preparation for Case Study . . . . .	189
7.4	Case Study Research Questions . . . . .	190
7.5	Units of Analysis . . . . .	192

7.6	Case Study General Descriptions & Rationale . . . . .	192
7.7	Case Study Results & Discussion . . . . .	196
7.7.1	RQ1: Applicability . . . . .	196
7.7.2	RQ2: Scalability . . . . .	197
7.7.3	RQ3: Efficiency . . . . .	198
7.7.4	RQ4: Effectiveness . . . . .	203
7.7.5	Compare FSMAApp with Other Approaches . . . . .	204
7.8	Threats to Validity . . . . .	212
<b>8</b>	<b>Future Work . . . . .</b>	<b>224</b>
8.1	Regression Testing . . . . .	224
8.2	New system domains . . . . .	224
8.3	Building Tools . . . . .	224
8.4	Effectiveness . . . . .	225
8.5	Improving Efficiency of Test Execution . . . . .	225
8.5.1	Efficiency Improvements During Execution . . . . .	225
8.5.2	Implementation . . . . .	229
<b>9</b>	<b>Conclusions . . . . .</b>	<b>233</b>
	<b>Appendix A Android Terms . . . . .</b>	<b>237</b>
	<b>Appendix B Family Medicines List App Screens . . . . .</b>	<b>238</b>
	<b>Appendix C BNF Grammar for Input Selection Constraints . . . . .</b>	<b>239</b>
	<b>Appendix D Family Medicines List App . . . . .</b>	<b>242</b>
D.1	Clusters and Nodes . . . . .	242
D.2	Reduced Test Sequences and Test Values . . . . .	245
D.3	Selenium Code of Test Paths . . . . .	252
	<b>Bibliography . . . . .</b>	<b>262</b>

## List of Tables

1.1	Publication . . . . .	5
2.1	Classification Framework [193] . . . . .	11
2.2	Studies Under Each Topic (Subcategory) [233] . . . . .	15
2.3	Manual and Automation Papers on Model building and Test Generation	16
2.4	Categories for Tools . . . . .	29
3.1	Constraints on Inputs . . . . .	39
3.2	FSMWeb Constraint of Typical Input Types . . . . .	39
3.3	Test Paths Through AFSM,FSM1,FSM2 . . . . .	42
3.4	Test Paths for $BM$ . . . . .	43
3.5	State-Event Matrix SE . . . . .	43
3.6	Potential Failure Scenarios $SP$ . . . . .	45
3.7	Mitigation Requirements . . . . .	47
3.8	Mitigation Patterns and Weaving Rules . . . . .	48
3.9	Selected $(i, p, e)$ Triplets and Resulting $FMT$ . . . . .	49
4.1	Software Characteristics of the Mortgage System [70] . . . . .	52
4.2	Decomposition of Mortgage System into Partitions . . . . .	55
4.3	Nodes for CD Form-FSM . . . . .	55
4.4	Input Constraints of Aggregate FSM for Mortgage System . . . . .	57
4.5	Input Constraints for LPD Cluster . . . . .	58
4.6	Input Constraint for Closing Documents (CD) FSM . . . . .	60
4.7	Test Sequence for Closing Documents (CD) FSM . . . . .	60
4.8	Test Paths of Aggregate FSM for the Mortgage System . . . . .	61
4.9	Test Paths of FSM for LPD Cluster . . . . .	61
4.10	Statistics of Tests Size . . . . .	62
4.11	Paths Generated by Substitution of $T_1$ . . . . .	62
4.12	Paths Generated by Substitution of $T_3$ . . . . .	63
4.13	Failure Types in Cluster Closing Documents (CD) . . . . .	64
4.14	Mitigation Requirement for CD Cluster . . . . .	65
4.15	State-Event Matrix for CD Cluster . . . . .	66
4.16	Abstract Test Paths (CD Cluster) . . . . .	66

4.17	Initial FMT for First Generation . . . . .	67
4.18	Effectiveness of GA . . . . .	68
4.19	FMT for Last Generation . . . . .	69
4.20	$f_{mt_i}$ that Found Defects . . . . .	70
4.21	Test Requirements and Mitigation Defects Found for Mortgage Lend- ing System . . . . .	70
4.22	Effort Comparison Between GA vs. Exhaustive Search . . . . .	72
4.23	The Size of $BT$ for CD vs. Mortgage System . . . . .	72
4.24	Classification of $BT$ After the Changes . . . . .	83
4.25	New State-Event Matrix $SE'$ . . . . .	84
4.26	New Potential Failure Scenarios $SP'$ . . . . .	85
4.27	$FMT''$ for Modified Model $BM'$ . . . . .	86
4.28	New State-Event Matrix $SE'$ . . . . .	87
4.29	The New Potential Failure Scenarios $SP'$ . . . . .	88
4.30	Selected Triplets and Constructing $FMT''$ for $F_{se}$ . . . . .	89
4.31	Reusable Tests After Deleting Failures $f_2$ and $f_3$ . . . . .	90
4.32	New State-Event Matrix $SE_a$ . . . . .	91
4.33	New Mitigation Requirements . . . . .	92
4.34	New Potential Failure Scenario $SP_a$ Due to the Added Failures $F_a$ . .	92
4.35	Selected Triplets and Resulting $FMT_{F_a}$ . . . . .	93
4.36	New State-Event Matrix $SE'$ . . . . .	95
4.37	New Potential Failure Scenarios $SP'_1$ ( $BM$ Change and $f_2, f_3$ Deleted) .	95
4.38	Constructing $FMT'$ with $PE'$ . . . . .	96
4.39	$PE_{WR}$ and Resulting $FMT$ . . . . .	100
4.40	Effect of Changes Using Selective Regression Testing (SR) and Full Retest (FR) . . . . .	102
4.41	Comparison of Approaches . . . . .	104
4.42	Selective Regression Testing vs. a Full New Regression Test Suite . .	109
5.1	Constraints on Inputs . . . . .	129
5.2	Components of Mobile Application (LAPs) . . . . .	133
5.3	Transitions of Figure 5.3 (Main Page Cluster AFSM) . . . . .	140
5.4	Transitions of Figure 5.4 (New Med Cluster) . . . . .	141
5.5	Transitions of Figure 5.5 (New Name Cluster) . . . . .	141
5.6	Transitions of Figure 5.6 (Modify Med Cluster) . . . . .	142
5.7	Transitions of Figure 5.7 (New NameP Cluster) . . . . .	142
5.8	Transitions of Figure 5.8 (New Medicines Cluster) . . . . .	143
5.9	Transitions of Figure 5.9 (Dosage Cluster) . . . . .	143
5.10	Transitions of Figure 5.10 (When Cluster) . . . . .	144
5.11	Transitions of Figure 5.11 (Instructions Cluster) . . . . .	144
5.12	Transitions for Figure 5.12 (Edit Cluster) . . . . .	145
5.13	Annotations for Main Page (AFSM) Transitions of Figure 5.3 . . . .	146

5.14	Main Page Test Sequences of Figure 5.3 . . . . .	147
5.15	New Med Cluster Test Sequences of Figure 5.4 . . . . .	147
5.16	New Name Cluster Test Sequence of Figure 5.5 . . . . .	147
5.17	New NameP Cluster Test Sequences of Figure 5.7 . . . . .	148
5.18	New Medicines Cluster Test Sequences of Figure 5.8 . . . . .	148
5.19	Dosage Cluster Test Sequences of Figure 5.9 . . . . .	148
5.20	When Cluster Test Sequences of Figure 5.10 . . . . .	148
5.21	Modify Med Cluster Test Sequences of Figure 5.6 . . . . .	148
5.22	Instructions Cluster Test Sequences of Figure 5.11 . . . . .	148
5.23	Edit Cluster Test Sequences of Figure 5.12 . . . . .	149
5.24	Aggregated Test Paths . . . . .	153
5.25	Test Path After Reduction Step . . . . .	155
5.26	Length of Before and After Reduction Step . . . . .	156
5.27	Test Path 8 With input values . . . . .	158
6.1	New Medicines (ESG) Decision Table . . . . .	165
6.2	New NameP (ESG) Decision Table . . . . .	166
6.3	Dosage (ESG) Decision Table . . . . .	166
6.4	When (ESG) Decision Table . . . . .	167
6.5	Instruction (ESG) Decision Table . . . . .	167
6.6	Edit (ESG) Decision Table . . . . .	168
6.7	Family Medicines List (ESG) Test Paths . . . . .	172
6.8	A GUI Crawling-Based Technique for Android Mobile Application Testing Edges . . . . .	181
6.9	Family Medicines List (GUI Crawling-Based Technique) Test Paths .	182
6.10	Parameter for GUI-Crawling-based Technique . . . . .	183
6.11	Comparison of FSMApp and ESG . . . . .	187
6.12	Comparison of Applying Techniques in Family Medicines List App .	188
7.1	Units of Analyses . . . . .	192
7.2	Android Mobile Applications . . . . .	196
7.4	Model Size Summary . . . . .	201
7.5	Summary of Test Generation and Execution . . . . .	202
7.6	Defect Summary . . . . .	203
7.3	Apps Components . . . . .	214
7.7	Comparison of Applying Techniques in Simple Calendar . . . . .	215
7.8	Comparison of Applying Techniques in Amaze File Manager . . . . .	216
7.9	Comparison of Applying Techniques in Timber . . . . .	217
7.10	Comparison of Applying Techniques in ML Manager . . . . .	218
7.11	Comparison of Applying Techniques in File Manager . . . . .	219
7.12	Comparison of Applying Techniques in Minimal ToDo . . . . .	220
7.13	Comparison of Applying Techniques in MIRAKEL: Task Management	221
7.14	Comparison of Applying Techniques in Memory Game Application .	222



7.15	Comparison of Applying Techniques in Todo List . . . . .	223
8.1	Test Paths for AFSM . . . . .	227
8.2	Save Execution . . . . .	231
A.1	Android Terms . . . . .	237
C.1	Grammar for Input Constraint Language . . . . .	241
D.1	Clusters and Nodes For Main . . . . .	242
D.2	Clusters and Nodes For New Med . . . . .	243
D.3	Clusters and Nodes For New Name . . . . .	243
D.4	Clusters and Nodes For Modify Med . . . . .	244
D.5	Clusters and Nodes For New NameP . . . . .	244
D.6	Clusters and Nodes For New Medicines . . . . .	244
D.7	Clusters and Nodes For Dosage . . . . .	244
D.8	Clusters and Nodes For When . . . . .	244
D.9	Clusters and Nodes For Instructions . . . . .	245
D.10	Clusters and Nodes For Edit . . . . .	245
D.11	Aggregated Test Paths Values . . . . .	245

## List of Figures

3.1	Three Optional Inputs, Any Order . . . . .	40
3.2	Behavioral Models $BM$ , $BM'$ . . . . .	42
3.3	Mitigation Models. . . . .	47
4.1	Mortgage System Logical View . . . . .	53
4.2	Aggregate FSMs with Partition and Top Level Navigation . . . . .	56
4.3	Loan Processing Data (LPD) Cluster . . . . .	56
4.4	FSM For Closing Documents (CD) Service . . . . .	59
4.5	Mitigation Model $MM_7$ . . . . .	64
4.6	Regression Testing Process . . . . .	76
4.7	Modified Mitigation Model $MM_4$ or $MM_6$ . . . . .	98
5.1	Process Mobile App . . . . .	119
5.2	Family Medicines List App . . . . .	121
5.3	Main Page . . . . .	123
5.4	Family Medicines List New Med . . . . .	123
5.5	Family Medicines List New Name . . . . .	124
5.6	Family Medicines List Modify Med . . . . .	124
5.7	Family Medicines List New NameP . . . . .	125
5.8	Family Medicines List New Medicines . . . . .	125
5.9	Family Medicines List Dosage . . . . .	126
5.10	Family Medicines List When . . . . .	126
5.11	Family Medicines List Instructions . . . . .	127
5.12	Family Medicines List Edit . . . . .	127
5.13	Annotated FSM for New Medicines Cluster of Table 5.8 . . . . .	139
5.14	Setup Connection . . . . .	160
5.15	Test 8 in Selenium . . . . .	161
5.16	Example Execution Results . . . . .	162
6.1	Family Medicines List (ESG) New Med . . . . .	169
6.2	Family Medicines List (ESG) Edit Med . . . . .	170
6.3	Family Medicines List (ESG) Add Patient Name . . . . .	171
6.4	A GUI Crawling-Based Technique Model . . . . .	178

6.5	A GUI Crawling-Based Technique Model . . . . .	179
6.6	A GUI Crawling-Based Technique Model . . . . .	180
7.1	Generation Time vs Number of Edges . . . . .	198
7.2	Generation Time vs Number of Test Sequences . . . . .	199
7.3	Input Selection Time vs Inputs and Actions . . . . .	199
7.4	Execution/Validation Time vs Test LOC . . . . .	199
7.5	Generation Time vs number of Edges . . . . .	206
7.6	Generation Time vs Number of Test Sequences . . . . .	206
7.7	Input Selection Time vs Inputs and Actions . . . . .	207
7.8	Execution/validation Time vs Test LOC . . . . .	207
8.1	FSMs Example . . . . .	226
8.2	Commonality Subpaths . . . . .	230
B.1	Family Medicines List App Screens . . . . .	238
D.1	Test 1 in Selenium . . . . .	252
D.2	Test 2 in Selenium . . . . .	253
D.3	Test 3 in Selenium . . . . .	254
D.4	Test 4 in Selenium . . . . .	255
D.5	Test 5 in Selenium . . . . .	256
D.6	Test 6 in Selenium . . . . .	257
D.7	Test 7 in Selenium . . . . .	258
D.8	Test 9 in Selenium . . . . .	259
D.9	Test 10 in Selenium . . . . .	260
D.10	Test 11 in Selenium . . . . .	261

# Chapter 1

## Introduction

### 1.1 Problem

A Mobile Application, or App, refers to software run on mobile phones or smart devices. Millions of Apps are available via App stores like Google Play<sup>1</sup> and Apple App<sup>2</sup> Store [135]. The revenues from Mobile Apps are projected to reach \$188.9 billion in 2020 [5]. The pervasiveness of App use also means that quality becomes a major concern. Fierce competition [204] means that a reliable App will be more successful.

While Apps share common technology with other software, especially web applications, they differ from desktop software in some important ways [224]:

- Interaction with other applications,
- Sensor handling such as touch screens and cameras,
- Both native and mobile web applications,
- A multitude of hardware devices and platforms,

---

<sup>1</sup><https://play.google.com/store?lil=en>

<sup>2</sup><https://itunes.apple.com/us/genre/ios/id36?mt=8>

- Heightened security concerns,
- Usability that is influenced by other Apps and by the common small size of the smart phone,
- Power consumption,
- Complexity of testing.

The complexity of testing arises from the fact that, in addition to the same issues as found in web applications, App testing must deal with issues related to mobility, transmission through software, and the issues listed above. Testing mobile Apps is clearly more complex than testing desktop applications.

Muccini et al. [169] investigated how mobile App testing differs from testing traditional applications. Mobile connectivity needs to be tested for different connectivity scenarios, networks, resource usage and associated performance degradation possibly resulting in incorrect system functioning. All of 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, as it can lead to computational explosion. Performance assessment is crucial. Many of these testing needs require that a functional test be executed for a number of specific environmental scenarios, set-ups, and devices.

This is one reason why test automation is clearly desirable and has been pursued quite successfully [66, 40, 164, 228, 39, 43, 123, 115, 221]. In most cases, the tools are not based on a model-based testing approach and still require the development of a test suite up front. They capture test inputs and play them back, or simply automate existing tests for different configurations, devices, and platforms.

Our interest is in the model-based black-box testing of mobile applications. Specifically, we are interested in extending an existing technique, FSMWeb [51] so we can apply it to test mobile Apps. In the cause of investigating this, we also realized that existing work around the FSMWeb body of knowledge needed improvements.

## 1.2 Existing Body of Work Related to FSMWeb

FSMWeb [51, 49, 139, 185] is a widely cited approach that tests web applications. Andrews et al. [51] proposed FSMWeb as a black-box model-based testing approach. The model consists of a hierarchical collection of FSMs. In addition, Andrews et al. [50] study the scalability issues of traditional FSMs of web applications compared to FSMWeb. FSMWeb compresses inputs using a special purpose input constraint language [50] reducing the model by as much as 90% . The case studies [50] show that FSMWeb is more efficient than conventional FSM techniques. Ran et al. [186] defined input selection for FSMWeb. They build two databases: a test database whose values are consumed during testing and the application database of the system under the test. Andrews et al. [49, 45] also propose an approach for selective regression testing of web application using FSMWeb and develop a cost-benefit tradeoff framework between brute force and selective regression testing.

An external failure is an undesirable event that affects system operation [141]. Examples include hardware failures, sensor failures, or network outages. External failures are not a result of faults in the software. In web applications, such failures can result in losses of millions of dollars, for example, when in 2006, Walmart.com was inaccessible for 10 hours during the holiday season [182]. Boukhris [68] provides an approach that systematically tests external failure mitigation for web applications by extending FSMWeb. He generates mitigation test requirements using a Genetic

Algorithm (GA) and proposes regressing testing for fail-safe testing. However, there are a number of issues that led to the need for improvements which we address a part of this dissertation:

- Lack of proper case study methodology.
- Lack of defining reusable test requirements for regression testing leading to major inefficiencies.
- Use of GA where search space is too small.

These will be addressed in this dissertation, in addition to the major topic, i.e. Black-Box Model-Based testing of Mobile Apps.

### 1.3 Research Agenda

Our main objective was to extend FSMWeb to test mobile Apps. However, when we reviewed the existing body of work related to FSMWeb, we realized that a number of improvements needed to be made, specifically:

- Show scalability with a larger case study: The approach to fail-safe testing [68] needed a larger formal case study that meets the methodology requirements of [190]. This case study is presented in Chapter 4.
- Make regression testing more efficient: The regression testing approach in [68] did not address reusable test requirements, considering them retestable and hence required a much larger number of regression tests than necessary. We provided an approach to classify test requirements as retestable, reusable and obsolete. This is discussed in Section 4.2.2.

Title	Year
Selective regression testing of safety-critical systems: a black box approach[47]	2015
A case study of black box fail-safe testing in web applications[70]	2017
A Comparison of Strategies to Generate Test Requirements for Fail-Safe Behavior[69]	2017
Black-Box model-Based regression testing of fail-Safe behavior in web applications[46]	2019

Table 1.1: Publication

- Large case study for regression testing: The case study for fail-safe regression testing in [68] did not meet case study methodology requirements. It also used genetic algorithms to generate test requirements. According to our comparison [69] of using coverage criteria vs. genetic algorithm to generate test requirements, this is not effective for the small amount of change in this case study. We completely redid the case study using coverage criteria and performed a comparison of the approaches. This is discussed in Section 4.1. We also added formal case study research questions, analyzed results to answer them, discussed threats to validity, and practical considerations.
- Improve performance: When reviewing test path generation, we noticed that many of them had initial subpaths in common which we used to improve performance. Likewise, parallel execution could reduce overall execution time. This is explored in future work Section 8.5.

Table 1.1 shows the published papers of this dissertation. Table 1.1 shows the title of the paper and the publication year. We published two journal papers and two conference papers.

This document is organized as follows: Chapter 2 describes related work in testing mobile apps, black-box testing approaches, regression testing approaches, and



functional of testing of web applications, as well as fault and failure taxonomies for web applications. Chapter 3 details the original FSMWeb approach as well as its extensions for fail-safe testing of web applications. Chapter 4 presents the improvements for fail-safe testing of web applications, including regression testing and a formal case study. Chapter 5 presents the extensions to the FSMWeb approach for testing mobile application (FSMApp). Chapter 6 compares FSMApp to several other approaches for MBT for mobile apps [88, 37] using one small example app. Chapter 7 describes a number of case studies to compare FSMApp with other approaches and to explore applicability, scalability, effectiveness and efficiency. Chapter 8 suggests further work. Chapter 9 draws conclusions.

## Chapter 2

### Background

The background chapter first summarizes Black-Box Model-Based Testing (MBT) techniques to test web applications; then it explores existing work for testing mobile Apps. Since we are also interested in regression testing, relevant work is presented next. Finally, since our improvements to FSMWeb include fail-safe testing, we present existing work in this area.

#### 2.1 Black-Box Model-Based Testing

Nguyen et al. [176] define Model-Based Testing (MBT) as an approach to generate test cases using an abstraction of the system under test (SUT). The model provides an abstract view of the SUT by focusing on particular system characteristics. Utting et al. [211] provide a survey on MBT. They define six dimensions of MBT approaches: model scope, characteristics, paradigm, test selection criteria, test generation technology, and test execution. Dias-Neto et al. [92] characterize 219 MBT techniques after analyzing 271 MBT papers and describe approaches that support the selection of MBT techniques for software projects, including risk factors. Risk factors may influence the use of these techniques in industry.

Utting et al. [211] classify MBT by notation used, such as State-Based, History-Based, Functional, Operational, Stochastic, and Transition-Based. Transition-Based notations are graphical node-and-arc notations that focus on defining the transitions between states of the system, such as Finite State Machines (FSMs). Transition-Based notations also include UML behavioral models, such as activity diagrams, sequence, and interaction diagrams [211]. This research addresses the use of MBT for functional testing of mobile applications. We apply MBT and utilize a hierarchical finite state machine model as described in Chapter 5.

Researchers provide many MBT techniques for web applications such as [166, 160, 98, 188, 142, 180, 103, 177, 51], including regression testing, for example [49, 139, 185, 207, 165, 138, 159, 95]. Andrews et al. [51] propose an approach to test web applications with Finite State Machines (FSMWeb). The approach is based on a black box functional model. FSMWeb is a hierarchical collection of FSMs. The approach consists of two phases: (1) building a model of the web application and (2) generating a test suite from the model. The first phase is completed in four steps: an application is divided into clusters, logical web pages are defined, FSMs are built for each cluster, and for the application (top) level [51]. The second phase is completed in three steps: (1) test paths for each cluster are generated by a variety of coverage criteria, such as edge coverage, (2) path aggregation generates abstract test paths, and (3) inputs are selected for the abstract test paths.

Andrews et al. [50] study the scalability issues of the traditional FSM of web applications compared to FSMWeb. The case studies [50] show that FSMWeb is more efficient than traditional FSM techniques. Ran et al. [186] defined algorithmic input selection for FSMWeb. They build two databases: a test database whose values are consumed during testing and the application database of the system under the test.

## 2.2 Testing Mobile Apps

Our main interest in testing Mobile Apps is Black-Box functional testing of mobile apps. As such, we are not interested in other testing activities for Mobile Apps, such as usability testing, configuration testing, exception handling testing, etc. This is reflected in how we review the literature. We first survey approaches for MBT for testing Mobile Apps. Then, we quickly review major approaches for test automation, primarily to determine what options we have to turn the tests generated by an MBT approach into executable tests.

### 2.2.1 MBT and App Testing

Sahinoglu et al. [193] present a mapping study of testing mobile applications. Their paper studies the research issues in mobile application testing and the most frequent test type and test level of available studies in mobile testing. They categorize existing studies into test levels, test types, and research issues. Test levels include system testing, acceptance testing, unit testing, component testing, and integration testing. Test types include compatibility, concurrency testing, conformance testing, performance testing, security testing, and usability testing<sup>1</sup>. The research issues discuss test execution automation, test case generation, test environments, cloud testing, and model-based testing. The mapping study concludes functional testing and usability testing are the most researched mobile testing areas. Also, they classify the publications according to research issues (Model-Based Testing, Static Code Analyze, Test Case Generation, Test Environment Management, Test Execu-

---

<sup>1</sup>These are outside the scope of our work.

tion Automation, and Testing in the Cloud<sup>2</sup>). There is only six model-based testing studies. This lack of MBT techniques shows that our proposed work is needed. Sahinoglu et al. [193] studied 123 papers on testing mobile applications, and the paper presents the count of every classification of test levels, test types and research issues of each study in the paper, but they did not discuss or reference the papers themselves. Table 2.1 defines the classification of the mapping study. Column one shows the category of the study and column two shows subcategories for each category. Column three defines what the subcategories are as defined in [193]. We added reference [42] on testing levels to Table 2.1. The definition of Model-based testing in Table 2.1 is not the same as the commonly used definition of MBT (see for example [45]). Here, we focus on the use of a behavioral model such as state transition diagram to generate tests.

Another systematic mapping study by Mendez-Porras et al. [164] discusses 83 empirical studies of automated testing of mobile applications. The paper shows the challenges and approaches of automated testing of mobile applications. Mendez-Porras et al. [164] address the variety of context events, fragmentation<sup>3</sup> in both software and hardware, and resource limitations of the mobile device. We are not focused on the fragmentation in both software and hardware, but we are interested in the different types of events: user GUI input such as a keyboard, physical context events such as GPS or Bluetooth, and social events like Facebook friends [151]. Also, the paper reports that the most commonly used method for empirical validation is a case study. The mapping study presents the main approaches for automated testing of mobile applications and the primary research of testing mobile applications

---

<sup>2</sup>We are interested in MBT only.

<sup>3</sup>Fragmentation means that Apps behavior is different on different android devices[111].

Table 2.1: Classification Framework [193]

Category	Subcategory	Definition
Test Levels	System Testing	“Access software with respect to architectural design and overall behavior.” [42]
	Acceptance Testing	“Assess software with respect to requirements or users’ needs.” [42]
	Unit Testing	“Assess software with respect to implementation.” [42]
	Component Testing	“Assess software with respect to detailed design.” [42]
	Integration Testing	“Assess software with respect to subsystem design.” [42]
Test Types	Compatibility	“The ability of two or more systems or components to perform their required functions while sharing the same hardware or software environment.” [104]
	Concurrency Testing	“Testing to determine how the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution, is handled by the component or system.” [104]
	Conformance Testing	“Conformance testing is testing to see if an implementation meets the requirements of a standard or specification.” [20]
	Performance Testing	“Testing conducted to evaluate the compliance of a system or component with specified performance requirements.” [104]
	Security Testing	“Testing to determine the security of the software product.” [12]
	Usability Testing	“Usability testing refers to evaluating software by testing it with representative users. Typically users will attempt to complete typical tasks while observers watch, listen and takes notes.” [12]
Research Issues	Test Execution Automation	“The use of software, e.g. capture/playback tools, to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and reporting functions.” [12]
	Test Case Generation	“A computational method for identifying test cases from data, logical relationships or other software requirements information.” [2]
	Test Environment	“An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.” [104]
	Cloud Testing	“Cloud Testing uses cloud infrastructure for software testing. Cloud computing offers use of virtualized hardware, effectively unlimited storage, and software services that can aid in reducing the execution time of large test suites in a cost-effective manner.” [209]
	Model-based Testing	“Testing based on a model of the component or system under test, e.g., reliability growth models, usage models such as operational profiles or behavioral models such as decision table or state transition diagram.” [12]

that have been validated using empirical studies<sup>4</sup>. It also analyzes the usefulness and accuracy of automated testing of mobile applications. They classify the empirical studies as follows: model-based testing, capture/replay, model-learning testing, systematic testing, fuzz testing, random testing, and script based testing. The definition of the categories is as follows:

- Model-based testing builds a model of the application under test and uses this model to generate test cases [176].
- Capture/replay captures events while the test cases are executed and then replays them automatically.
- Model-learning testing builds a model of the GUI application in conjunction with a testing engine and guides the generation of user input sequences based on the model. Depending on the quality of the tests used or generated by the test engine, the model may be incomplete.
- Systematic testing automatically and systematically generates input events to exercise applications. It is usually based on test criteria and an abstraction of the software, hence has overlap with MBT. The systematic testing applies symbolic execution [77, 105, 133] to generate inputs for Android apps. Symbolic execution automatically partitions the domain of inputs such that each partition corresponds to a unique program behavior while avoiding unnecessary inputs [157]. We are not interested in symbolic execution because it is not black-box testing.

---

<sup>4</sup>This is narrower than our scope, since we also consider approaches that still lack empirical validation.

- Fuzz testing generates a large number of simple inputs to applications. Fuzz testing is a black-box approach and fully automatic to generate random UI events for to Android apps in a mobile device emulator. Fuzz testing does not generate inputs that represent typical user action sequences such as playing a game. The tool for Fuzz testing is Monkey [32]. We did not consider this because FSMAApp models black-box user behavior.
- Random testing generates random sequences of user input events for the application being tested.
- Script based testing requires manually writing test cases in the script language. These are then transformed automatically into executable tests and executed.

They conclude that 40% of the studies use GUI-based models of the application and 30% of the approaches use model-based testing. This dissertation focuses on Black-Box MBT of mobile applications. Hence, only the first category is of relevant to the scope of this dissertation. Mendez-Porrás et al. [164] cite [228, 37, 232, 91, 123, 221, 38, 194, 128, 55, 198, 102] as model-based testing papers. We exclude a number of papers from this list as they are not in our scope. This subsection already discussed [37, 128, 198]. This subsection does not discuss white-box testing [91, 123] and grey-box testing [228]. We are also not interested in testing security [194], data-flow analysis [55] and life cycle testing of the application [102] Zaeem et al. [232] is not discussed in this dissertation because it is related to oracle problems. Wang et al. [221] identify several difficulties for automating GUI testing and study the high cost for achieving coverage of the traversal algorithm to generate a GUI tree model. It extends the crawling technique in Amalfitano et al. [37] to increase GUI coverage. Also, Amalfitano et al. [38] present a Event-Based Testing approach that Mendez-Porrás et al. [164] classify as MBT.



This dissertation focuses on creating the model for testing mobile application and we discuss Amalfitano et al. [37] later as one of the approaches that we will compare to FSMAApp.

Several research papers are not included in Mendez-Porras et al. [164]. Zein et al. [233] present another mapping study of mobile application testing techniques. The goal of the mapping study is based on the classification of empirical studies. The main research question is what are the empirical studies that investigate mobile application testing techniques and what are the challenges? Zein et al. [233] consider studies of mobile testing techniques, services <sup>5</sup>, security and usability testing of mobile applications, and the challenges of testing mobile applications. The mapping study uses five categories: Usability testing, Test automation, Context-awareness, Security, and a general category. The general category includes all the studies which are not in the other areas. The general category does not focus on usability testing, test automation, context-awareness, security, performance testing [75, 132] and compatibility testing of the mobile application with the underlying operating systems [218, 219, 236]. Table 2.2 lists [58, 65, 99, 187, 61, 145, 212, 152, 183, 118, 184, 100, 137, 63, 62, 78, 67, 161, 225] as usability testing, [37, 175, 97, 199, 66, 39, 240, 130, 184, 153, 201, 235, 87, 168, 206, 55, 148, 84, 116, 220, 226, 239, 88, 36, 33, 124, 117, 40, 107] as test automation, [223, 38, 195, 125, 192, 222, 231, 215] as context-awareness, [129, 194, 155, 53, 80, 108, 134, 114] as security testing and [151, 75, 132, 101, 150, 102, 90, 91, 213, 202, 218, 236, 237, 219, 35] as general category. Column one shows the category name and column two presents the references. Column three shows the number of papers for each category. None of the categories defined in this mapping study directly speaks to the scope of this dissertation.

---

<sup>5</sup>"Mobile services are currently targeting time and safety critical contexts such as abnormal and disaster management situations"[233].

However, the category Test Automation contains two potentially relevant subcategories: Model-based test automation and Black-box test automation. The other categories are not relevant to our scope.

Table 2.2: Studies Under Each Topic (Subcategory) [233]

Category	Studies	Total # studies
Usability testing	[58], [65], [99], [187], [61], [145], [212], [152], [183], [118], [184], [100], [137], [63], [62], [78], [67], [161], [225]	19
Test automation	[37], [175], [97], [199], [66], [39], [240], [130], [184], [153], [201], [235], [87], [168], [206], [55], [148], [84], [116], [220], [226], [239], [88], [36], [33], [124], [117], [40], [107]	29
Context-awareness	[223], [38], [195], [125], [192], [222], [231], [215]	8
Security testing	[129], [194], [155], [53], [80], [108], [134], [114]	8
General category	[151], [75], [132], [101], [150], [102], [90], [91], [213], [202], [218], [236], [237], [219], [35]	15

Test automation has the following subcategories: Data, Portable operating system libraries with knowledge and reasoning, Sensitive-events, Scripted user interface, Exhaustive test amplification, Reverse engineering, Static taint-style data flow analysis, Depth-first exploration, Contextual fuzzing, Machine learning, Approximate execution, Automated mobile testing as a service, Parallel GUI testing using a master-slave model, Search, Systematic exploration of test suites, Sensor and event-stream based approach, and Sensor simulation<sup>6</sup>. Zein et al. [233] classify [37, 39, 153, 87, 206, 88, 124, 40] as Model-based testing methods. Not all are relevant to this dissertation.

---

<sup>6</sup>Zein et al. [233] classify test automation into too many categories. Most of them have only one research paper.

We exclude [40] because it is combination of machine-learning and model-based testing and we are interested in Model-based testing only. Also, [206] focuses on testing environments such as connection to the wireless network. [124] is a white-box testing approach.

Examining the papers listed under Model-based testing and Black-box testing yielded the following papers that fall into our scope [37, 39, 153, 87, 206, 88, 124, 40, 57].

Since we are interested in MBT and Black-box testing we categorize test automation into Model-Based testing, Black-Box Testing, and Testing Approach. The Model-based testing approach builds a model of the application being tested and uses this model to generate tests. Types of MBT: (1) The user will create the model manually, (2) The tool will generate the model automatically, (3) The user generates the test manually, (4) The tools generate tests and execute them. This dissertation focuses on model-based testing black-box testing of Mobile Apps, whether or not the steps are automated.

Table 2.3 summarizes research that use manual or automated model building or test generation. [128, 153, 88] build the model manually and [37, 39, 87, 124, 40, 205, 57] build the model automatically. All methods generate the test case automatically. This dissertation compares FSMAApp with [88, 37] because they describe the generation of the model in sufficient detail. First, we will describe the MBT methods using manual model generation, then those that use an automated method.

Table 2.3: Manual and Automation Papers on Model building and Test Generation

	Model-building	Test generation
Manual	[128], [88]	
Automated	[37], [153], [39], [87], [205], [57]	[128], [153], [88], [37], [39], [87], [205], [57]

Jing et al. [128] present a Model-based Conformance Testing Framework (MCTF) for Android Applications. Testers need to generate the model manually from the requirements. MCTF consists of four steps: System Modeling, Test Case Generation, Test Case Translation, and Test Case Execution. System Modeling derives the parameters and properties of an App, while Test Case Generation generates abstract tests automatically with Alloy Analyzer. Alloy Analyzer is 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 operating system is then tested with the packages. Packages are defined as test cases generated by Alloy Analyzer. The packages are run by Android’s Instrumentation Test Runner. They can access the Android’s applications. The model is generated from requirement documents.

Lu et al. [153] propose an activity page based model to automate functional testing for mobile applications. An activity page based model optimizes (reduces) the code of the crawling algorithm to generate a test case. The Activity page based model is a directed graph. An activity page is thus similar to a screen. It models mobile activity as a state. Edges represent a trigger event between the states. The mobile activity is the activity page with input components and related events. The model can be generated either by (1) the UIAutomation tool [31]. It provides the information about the activity page with all input components and events. or (2) Create a Model-based on an image comparison for every event. The Monkeyrunner tool [17] takes the screenshot of the activity page with the fired event. They generate test cases by applying a crawling algorithm. Lu et al. [153] present two modeling methods based on an activity page based model <sup>7</sup>. Android apps contain activity

---

<sup>7</sup>We focus on the activity page based model from GUI ignoring the method that uses the source code because our goal is black-box functional testing.

component which helps to develop the user interface of an app. An app usually contains one or more activity classes to provide GUI interfaces for the user. Lu et al. describe the following phases:

1. Create the activity page based model. The activity page is described as a tuple  $\langle \text{activity page, event, visible component, properties of each components, value component} \rangle$ . This tuple is similar to the input-action constraints in FSMApp and the decision tables in the ESG method. The main user interface is considered as the first activity page. All descriptions of the activity page are collected. Then, the events are fired to capture all possible activity pages which are accessible from the first activity page. This process continues until all activity pages have been reached. The input selection is generated ad-hoc and recorded on the edge.
2. The test cases are generated by the following crawling algorithm:
  - (a) Step 1: Get all nodes that connect to the "exit" node of the activity page can be used as a test case to start up the rest of the activity pages (start-up nodes).
  - (b) Step 2: Take the main activity page (start node) through the other nodes from step1. Determine all the shortest paths from the start node to exit nodes.
  - (c) Step 3: Create all single-loop paths of each start-up activity page that start and end at the start-up activity page.
  - (d) Step 4: If there are uncovered edges by step 3, then calculate the shortest multi-loop path for each uncovered edge. In step 4, uncovered edges will be covered as complementary test cases.

3. Robotium framework executes the test paths and captures any failures.

It is not clear how and when the tuples are used and at which stage input is selected. The example on the paper does not help in that regard either. It is hard to compare FSMAApp with this activity page based model because (1) the description of the input constraints is incomplete, especially related to dependencies between inputs and (2) it is unclear at which phase the input constraints are resolved into values. When we used the crawling algorithm to generate test paths, it also became clear that for the small example in chapter 6, the paths were inordinately large. The number of test paths is 62, consisting of 350 nodes. The execution time is 29 minutes. FSMAApp generated 11 test paths with 45 nodes. The execution time is 11 minutes. FSMAApp needs 80% fewer test paths and 60% less execution time.

Amalfitano et al. [39] present another GUI automated technique to test Android apps. They implement the technique in a tool called AndroidRipper. The technique is based on a GUIRipper. This is a dynamic approach in which the software's GUI is automatically traversed by opening all its windows and extracting all their widgets (GUI Objects), properties, and values [163]. GUIRipping reverse engineers the GUI of the application. The ripping generates a tree model. The ripping technique is an iterative process with the following parameters: event, action, task, and GUI exploration criterion. An event is a user event, such as fill text-box, or an event related to activity classes like onStart function or events from other sources such as GPS. The action is a data input and event command such as <click>. The task consist of multiple actions, such as <click> where performs the action click. The GUI exploration criterion is a logical criterion to explore the GUI tree. The tool takes a long time to generate test cases for large apps, and it does not support some input, such as sensors, whereas FSMAApp tests large apps and includes more inputs.

Costa et al. [87] adopt Pattern-Based GUI Testing (PBGT) to test mobile apps. PBGT is based on User Interface Test Patterns [173] specifically developed to test web applications. Costa et al. [87] aim to increase the reusability and reduce the effort of modeling and testing mobile Apps. The differences between web and mobile versions of their PBGT approach is in the mapping and interaction strategy, and the fact that mobile apps can call other applications. PBGT has five main components:

1. PARADIGM-DSL is a domain specific language (DSL) for building GUI test models based on user interface test patterns (UITP).
2. PARADIGM-ME supports tester to build the test models of the application [167].
3. PARADIGM-TG is a tool to generate test cases from PARADIGM-ME model with different coverage criteria, such as edge coverage to cover all the paths from the start node to end node [172].
4. PARADIGM-TE is a tool to execute test cases and analyze the coverage on the model and the code then create the test results reports [217].
5. PARADIGM-RE is an automated reverse engineering tool to create a model of an existent mobile application into PARADIGM-ME model tool then the designer can modify the model [174].

This approach does not include some gestures and components like swipe and zooming whereas FSMApp (Chapter 5) supports components. Also, it does not support varying screen size and loops. It needs a separate model for every function of the mobile app. They cannot be connected together to create a hierarchical model to test the mobile app. We therefore decided not to compare FSMApp with this approach using our case studies.

Costa et al. [87] use a domain specific language to model and test mobile applications. Because their annotation does not use a node and edge notation like FSMApp, it is difficult to compare the two approaches. Therefore, we decided not to include this method in our case study comparison.

Takala et al. [205] present a test automation solution for testing Android apps. They use Model-based testing tools (TEMA) [205]. The TEMA tool is a set of Model-based tools for different phases of MBT. The phases are modeling, design, generation and debugging tests. TEMA models are Labeled State Transition Systems (LSTS) [121, 122] which contain: state, transition, actions, and labels. The model can be divided into small components that are connected. Each component has two levels: an action machine and a refinement machine. The action machine explains the high-level functionality of the apps with words (action of transition) and state verification (state of SUT). The refinement machine describes action words and state verifications using a keyword. The keyword is an abstraction of a user action or state verification such as "press key" or "search text". The model is complex even for small apps because each component is a combination of action machine and refinement machine of each function of a mobile application. Also, the two levels make testing of large mobile apps difficult because a refinement machine state model can only connect to one action machine. In our approach, we implement hierarchical levels, and the clusters can be used for many states which mitigate the state explosion problem and simplify the generation of a model for large mobile apps.

Takala et al. [205] used TEMA tools with state machines to test mobile applications. They described the following phases:

1. Create a model with labeled state transition systems (LSTS). An LSTS has states, labels of the state, transitions between states, actions of the transitions.



The models can be divided into smaller model components to understand the model quickly and make it easier to maintain. The full model is the result of combining the components in a process called parallel composition. The parallel composition links specific actions of the models together. The component consists of two levels of separate state machines: an action machine and a refinement machine. The action machine describes high-level functionality with action words and state verification. An action word describes a small use case in the app on the transition such as saving a file. A state verification describes the state of the system under test and is used to verify that it corresponds to the state of the model as described in the state label. The refinement machine describes the implementation of the action words and state verification by keywords. A keyword is a defined combination of actions which describes the execution of the test case. The advantage of the two-layer model architecture is that the action machines can be reused but the refinement machine must be remodeled when they are used in a new content such as with other mobile devices. The TEMA model supports localization tables or data tables for the test data. A localization table contains the localized string. The localized string is chosen during test generation. The data tables store complex structured data like input given to the App under test.

2. The test generation can use online or offline approaches. The online approach generates tests while the SUT is tested with tests generated in earlier steps. Later, the tool generates the other test cases depending on the returned result to adapt test cases to different responses. The online approach allows random tests. The offline approach generates test suites from the model then executes them. The TEMA test generation tool uses the online approach.

3. The TEMA execution tool runs the test suite alongside the test generation (using the online approach).

Takala et al. [205] tested the BBC News Widget. It took a few days to generate the model using the TEMA tool. The tool generated 240 test cases with 27000 action words and 50000 keywords. The total time to execute the test cases was 115 hours with an average of about 30 min for each test case. We applied the approach to the todo app [30] because it is a small app. We created the model manually because the TEMA tool does not have clear documentation for installation and use. The model of the todo app [30] has 37 nodes, 61 edges, 1700 keywords, and 900 action keywords. We estimated that it would take four hours to run the test cases with the keywords. The number of test paths of this approach compared to FSMApp is very large because FSMApp only required 11 minutes for executing all test paths. It is complicated to generate the model and the keywords manually. For these reasons, we excluded Takala et al. [205] approach.

Baek et al. [57] present a Model-based Black-box approach to test Android apps with a set of multi-level GUI Comparison Criteria (GUICC). The approach creates a directed graph which contains ScreenNode and EventEdge. The ScreenNode is a GUI state which includes screen information to distinguish it from other ScreenNodes by performing a GUI Comparison. GUI state represents the type of GUI information: package name, activity name <sup>8</sup>, layout and executable of the widgets, and content information. GUICC has five levels. The first level compares the package name of the running screen to visited ScreenNodes of the GUI graph. If the package name is running in the background of the mobile device, then the approach applies the second level.

---

<sup>8</sup>See Appendix for more details.

The second level compares the activity names of the running screen with names of the activities in the GUI Graph. If the activity name is not found in the GUI graph, a new ScreenNode is created in the GUI Graph. The third level compares the layout of the widgets with GUI graph ScreenNodes: if they have the same package name and activity name. If the layout of the current widgets is different, then GUICC creates a new ScreenNode and connects it to the GUI graph. The fourth level compares the events of the widgets and the fifth level compares the content information. Baek et al. [57] develop a testing framework around GUICC. The framework has three modules: (1) The communication layer connects between desktop and mobile device. (2) The EventAgent is a tool to run the test on a mobile device. (3) The testing Engine generates a GUI graph with GUICC, test inputs, and test cases. The framework is not open to the public and Baek et al. [57] do not explain the approach in enough detail. Therefore we are not able to compare the approach with our approach.

de Cleve Farto et al. [88] evaluate the use of MBT to verify and validate mobile applications 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 one of the modeling techniques used in MBT [88]. ESG is a directed graph that represents the events (nodes) and possible sequences of events (edges). The testing approach has three phases (1) Create the ESG model, (2) Generate and implement test cases from the ESG model and (3) Execute the test cases with Robotium and collect data. de Cleve Farto et al. [88] generate test paths from the start node to the end node covering all edges. The dissertation [59] shows an example with multiple test paths. Their tool determines how many paths are generated. It does not allow for any other coverage criteria.

This approach does not provide input information as part of the model such as input constraints but they consider them separately in a decision table. There are no coverage criteria for combinations from the decision table, such as defined in [64]. Chapter 6 describes and compares this approach with FSMApp in detail with an example.

Amalfitano et al. [37] present a technique and a tool for crash testing and regression testing for Android apps. Amalfitano et al. [37] use a crawler technique to automatically build the model from the GUI and generate the test cases automatically. The model generates a GUI tree using an iterative depth-first search. The GUI tree is represented as nodes and edges. The nodes represent the user interfaces of the Android application, and the edges describe the event-based transition between the nodes. The GUI tree is built using an iterative algorithm which relies on two main temporary lists (for events and interfaces). The technique fires an event to capture the user interface (screen) and data of the interface. The tree is generated by considering the interface as a node and links the interfaces together until it reaches interface on for which the temporary interface's list is created (to avoid a cycle). After building the tree, test cases are generated. They also provide a tool for the technique named  $A^2T^2$  (Android Automatic Testing Tool). The tool generates the model and test cases. This technique generates a large number of test cases to test a small mobile application as represented in the example [37]. Chapter 6 will describe and compare this approach with FSMApp in detail with an example.

In summary, we will describe in detail the following approaches: [37, 88] and compare them to FSMApp as part of our case study validation.

### 2.2.2 General App Testing Tools

The previous section describes MBT techniques that may or may not also include tools for making tests executable and running tests. This section discusses tools that have been used in practice to build and execute test cases, regardless of whether they are designed using MBT or not.

We discuss these techniques because our case studies need to use such tools to execute tests. We are only interested in open source tools. Some tools generate tests based on a testing strategy, others do not. Choudhary et al. [85] compare the main mobile testing tools. They found 14 open source tools. They classify the tools into random, model-based and systematic strategy. Table 2.4 defines the three classifications and the differences between the model-based and systematic strategy. Tools that implement random test generation include Monkey [32], Dynodroid [157], DroidFuzzer [230], IntentFuzzer [196], and Null IntentFuzzer [18]. GUIRipper [39], ORBIT [228], *A<sup>3</sup>E – Depth – First* [55], SwiftHand [84], and PUMA [113] are Model-based strategy tools. The systematic strategy has four tools: *A<sup>3</sup>E – Targeted* [55], EvoDriod [158], ACTEvo [43], and JPF-Android [214]. Choudhary et al. [85] present a brief description of the tools and compares them by ease of use of the tool, Android framework compatibility, code coverage achieved and fault detection ability. This dissertation focuses on tools for model-based testing and test suite execution tools to test mobile applications. Therefore, we will not discuss the random and systematic tools. Also, ORBIT [228] is excluded because it is a grey-box testing tool that uses data-flow analysis [55]. SwiftHand [84] is a usability testing tool. PUMA [113] provides a programmable user interface automation framework for conducting dynamic analyses of mobile apps at scale. GUIRipper [39] was discussed in Subsection 2.2.1.

Lamsa [140] presents a comparison of 26 GUI testing tools. He concludes that five tools are popular based on a Google search for the tools using Google trends<sup>9</sup> and a computer science forum (Stack orverflow)<sup>10</sup>. He compares the five tools by ease developing test suites, execution time, amount of code, reliability, and compatibility with the mobile applications. These popular tools are Espresso [27], Tau [26], Appium [7], Robotium [22] and UIAutomator [31]. He concludes that Espresso is the best tool, Appium is in the middle, and the worst is UIAutomator.

Linares-Vasquez et al. [149] present a survey of frameworks, tools, and services to test mobile applications. The tools are categorized into: automation frameworks & APIs, record & replay, automated GUI-input generation, bugs & error reporting monitoring tools, testing services, cloud testing services, and device streaming. Automated GUI-Input Generation tools have subcategory: Random-Based, systematic, Model-Based and others. The categories are defined in Table 2.4. Column one shows the category and subcategory of the tools. Column two defines the category, and column three shows the tools that are based on MBT Black-box testing, applying Finite State Machines to test mobile applications.

This dissertation is focused on MBT. The following are MBT tools:

Appium [7] is a test automation framework for testing native, hybrid, and mobile web apps. Appium works cross-platform and supports most programming languages, such as Java and Ruby. Appium also supports Jelly Bean (Android 4.2) or higher but has weak documentation. Appium is an open-source tool to execute text cases. Subsection 5.5 describes Appium in more detail.

Eggplant [8] is a black-box GUI test automation tool. Eggplant uses image recognition algorithms for the GUI object-level of the application. Eggplant can

---

<sup>9</sup><https://www.google.com/trends/>

<sup>10</sup><https://stackoverflow.com/>

identify images, texts, and record and replay, but only identifies web browser objects in the phone, but not native or web application objects. Eggplant is not a free tool.

Ranorex [21] is a black-box GUI test tool for testing desktop, web-based, and mobile applications. It uses a standard programming language, such as VB.Net, and Delphi, for scripting. The tester can customize the test report engine. It also can be used without a script. In this case, the user generates test cases while navigating through the app. It does not support all gestures, drag and drop operations, or launching of closed applications. Ranorex is not a free tool, hence out of our scope.

Robotium [22] is a test framework for Android applications. Robotium allows test case developers to write function and system test scenarios. Robotium provides user interaction such as clicking, touching, and any mobile application gestures. Robotium cannot handle web components, or more than one application, and it is slow to build test cases. This dissertation aims to provide a testing approach for all types of mobile applications (native, web-based and hybrid).

In summary, this leaves Appium as the tool to use in our case studies for making tests executable and running them.

Table 2.4: Categories for Tools

Category & subcategory	Definition	Tools
Automation APIs/Frameworks	It is a tool or framework to automate testing of mobile applications.	
Record and Replay Tools	It is a tool to run the test without programming knowledge.	
Automated Test Input Generation Techniques	It is a process to generate test cases automatically.	
Random Based Input Generation	It is a tool generates random test with independent inputs.	
Model-based Input Generation	The tool builds the model of the app under the test to generate test cases.	[7, 8, 39, 21, 22]
Systematic Input Generation	It is a tool to test a mobile app against a variety of input to find errors. It uses symbolic execution and evolutionary algorithms.[85]	
Other Input Generation	It includes programmable, scripting, symbolic, multiple and search-based mobile testing tools.	
Bug and Error Reporting/ Monitoring Tools	It is a tool for supporting bug reporting or monitoring crashes and resources consumption at run-time. [149].	
Test Services	The tools include: Crowd-sourced, usability testing, security testing, and localization testing.	
Cloud Testing Services	Tools to test the mobile application by cloud services.	
Device Streaming Tools	Tools for Device Streaming can facilitate the mobile testing process by allowing a developer to mirror a connected device to their personal PC, or access devices remotely over the internet [149].	



## 2.3 Regression Testing (RT)

The purpose of regression testing is to check that new or modified functionality works, to ensure the continuous working of the unmodified parts of the software, and to validate that the modified software as a whole functions correctly [146]. There are two main approaches for regression testing: Retest All and Selective Regression Testing. The Retest-All approach simply tests the system all over again. This is appropriate when changes affect most of the software, but inefficient when only selected parts of the software changed. Selective Regression testing considers the modifications that affect parts of the software and only tests those. Selective Regression Testing may reduce the time and cost of retesting the software. Rothermel and Harrold [189] define a selective regression testing process as follows:

1. Identify changes that affect the software.
2. Determine obsolete test cases that are not valid for the new software version and remove them from tests set  $T$ , resulting in  $T'$ .
3. Execute  $T'$ .
4. Generate new test cases  $T''$  for the modified or new part of the software that are not tested with  $T'$ .
5. Execute  $T''$ .

Steps 2 and 3 test that the modifications have not broken any existing functions. Steps 4 and 5 test that the modifications and new code work correctly.  $T'$  is determined based on the existing test cases and the regression testing approaches. Leung and White [147] group the existing tests case into five categories: reusable,

retestable, obsolete, new-structural and new-specification test cases. A reusable test case tests an unmodified function and does not need to be executed. A retestable test case tests the modified part of the software and needs to be executed. An obsolete test case is no longer valid. A new-structural test case tests the changed software structure. A new-specification test case tests the functionality of the modified specification for the software.

Chen *et al.* [82] characterize regression tests: A Targeted Test tests functionality of the changed parts of the software, and a Safety Test addresses risks. Chen *et al.* [82] generate test cases using UML activity diagrams. They classify two types of changes; code changes and system behavior changes.

Orso *et al.* [178] present an approach similar to Chen *et al.* [82], but they apply a statechart diagram instead of UML activity diagrams. Their approach first compares between the new and the old versions of the statecharts and identifies test cases that are affected by the changes. Briand *et al.* [72] [74] present a regression testing technique using use case, sequence, and class diagrams. They provide definitions for 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.* [120] introduce a regression test selection method using UML state machines and class diagrams.

Regression testing can be very costly, hence Test-suite reduction techniques are used to reduce the number of test cases. Korel *et al.* [136] propose a requirement-based regression test suite reduction approach. They use an EFSM model to reduce the size of a regression test suite. Chen *et al.* [83] extend the work of Korel *et al.* [136] but propose a reduction approach in terms of modified transitions and dependencies between transitions. Andrews *et al.* [49] present a black-box regression testing technique for FSMWeb. Furthermore, Andrews *et al.* [45] study and com-

pare a cost-benefit tradeoff framework between brute force and selective regression testing. Their framework considers various cost factors related to steps in regression testing. FSMWeb does not address testing Mobile Apps because it cannot model all possible screens and actions in mobile Apps. Our goal is to extend FSMWeb to enable testing of mobile Apps and to partially automate the testing process. Also, none of these approaches consider fail-safe testing with regression testing or testing of mobile Apps.

## 2.4 Fail-Safe Testing

This type of testing is interested in external failures and their proper mitigation. An external failure is an undesirable event that affects system operation [141]. Examples of external failures are a database crash, loss of a network connection, or unavailability of a server on which the software depends. External failures are not a result of faults in the software.

Many researchers try to classify external failures or a fault taxonomy. Some types of the failures or faults required mitigation actions. Pertet and Narasimhan [182] study the causes and effects of web failures. They show cause of the affects of most failures: software failures, human/operator errors, hardware/environmental failures, and security violations. They show the failures: unavailable systems, exceptions, access violations, incorrect answers, data loss and corruption, and poor performance. Ardagna et al. [52] classify web fault types into infrastructure and middle-ware faults, web service faults, and web application faults. Additionally, they define recovery actions as retry services, substitute services, missing parameter completion, service reallocation and changing of process structure. Ma and Tian [156] classify web failures as host, network, or browser failures, source or content failures, and

user errors.

Guo and Sampath [109] study logic faults and compatibility faults and they classify browser interaction faults, session faults, paging faults, server-side parsing faults under Logic faults. All the papers are great for classifying web faults and failures, only Ardagna et al. ([52]) consider both failures and recovery.

Zeng et al. [234] propose recovery in the form of exception management in term of web services. They study the application and process exceptions. A service can be affected by a time delay or experience a degradation of service quality. They classify the types of recovery action as retry, skip, replace, try alternative, compensate, and timeout. Lu et al. [154] define the exceptions and exception handling policies formally for state charts. They define mitigation policies: skip, abort, retry, try alternative, compensate, replace, and timeout. Finally, Brambilla et al. [71] categorize exceptions as user-generated, application generated or infrastructure related. They defined the policies of exception handling as accept, reject, abort, ignore, and resume. Cabral and Marques [76] analyze frequently used exception handler categories for Java and .net applications: empty, log, alternative, throw, continue, return, rollback, close, assert, delegates, and an "other" category which includes action types that do not correspond to any of the prior categories.

Avizienis et al. [54] present a taxonomy for fault handling such as rollback, rollforward, and compensation. Ye et al. [229] provide a systematic approach for the selection of suitable mitigation strategies based on a taxonomy of failure types. Their approach confers the use of an unreliable component in the critical application with increased confidence. There is no one of the paper study the mitigation with Model-Based Testing. Furthermore, more formal exception handling patterns have been described for process and Work-flow models [144, 110, 171, 106, 210]. Jiang et al. [127] build an exception control flow graph. The graph uses data flow coverage

criteria on variables that are related to the exception. Sinha et al. [200] also investigate white-box testing with exception handling by using control flow and data flow analysis. Andrews *et al.* [44] propose a method to enhance an existing MBT technique, FSMWeb [51] that leverages a black box test suite derived from the model and transforms it into a series of tests for various failures that test their proper mitigation. These tests are called Failure Mitigation Tests (FMT). This approach also has the advantage that one can proceed with functional testing as usual, i.e. existing test suites do not have to be regenerated, failures can be injected at selected points in the existing test suite, and a mitigation test is created by modifying the functional test at the point of failure with required mitigation behavior.

Many papers classify failure type of web applications [156, 182, 52], but there is little in terms of categories treating external failures of mobile apps. Cinque et al. [86] present a measurement based failure characterization for Mobile apps. They classify failures as: (1) Freeze when a device does not respond to the user input, or the device output becomes constant. (2) Self-shutdown when the device turns off by itself. (3) Unstable behavior when the device changes behavior without user input such as backlight flashing. (4) Output failure when the device's output differs from the configuration such as a different ring. (5) Input failure such as when a user input has no effect on the smartphone (e.g. when a soft keyboard input does not work). Also, Cinque et al. [86] classify user-initiated recovery as: (1) Repeat the user action until the device responds. (2) Wait an amount of time for the device to respond. (3) Reboot the device. (4) Remove battery (this is mainly performed when the device freezes; however, in newer phones the user cannot remove the battery easily). (5) Service at the phone center.

Vijayalakshmi [216] analyzes the Android operating system by using failure mode and effect analysis [208]. Vijayalakshmi [216] categorizes failures into hardware and

software. Vijayalakshmi uses the same software failures categories from Cinque et al. [86]. The hardware failure modes are: (1) Keyboard does not work (e.g. the phone fell into the water) (2) Battery Failure due to inappropriate use of the battery or overcharging. (3) Mobile phone breaks due to low quality manufacturing (e.g. the user should be careful not to drop the phone, otherwise it will break. (4) Screen quality can affect the system such as color change or low image quality. (5) Manufacturing Error of power supply unit. Since fail-safe testing of mobile apps is outside of our scope, we are only concerned with fail-safe testing related to using FSM, and the case study and regression testing approach described in Chapter 4.

## Chapter 3

### Original Approach

#### 3.1 Testing Fail-Safe Behavior w/FSM Web

This section describes the approach of testing fail-safe behavior for which we develop a selective regression testing approach (section 3.1.1), similar to Boukhris *et al.* [70]. It differs in how test requirements are generated.

##### 3.1.1 Process

The black-box test generation process for testing fail-safe behavior consists of the following steps:

1. Generate test cases from the behavioral model (section 3.2).
2. Identify failure events and their mitigation (section 3.3).
3. Generate mitigation tests from the mitigation models (section 3.6).
4. Generate test requirements using coverage criteria (section 3.5).
5. Generate abstract test paths for each test requirement by weaving mitigations into the primary test path using weaving rules (section 3.7).

6. Generate, execute and validate tests (section 3.8).

The failure mitigation test process uses FSMWeb [51] black box testing of required functionality, and its associated behavioral testing criteria ( $BC$ ), and resulting behavioral test paths ( $BT$ ). System requirements identify types of failure events and any required mitigation actions. This is used to build failure mitigation models ( $MM$ ) for which mitigation coverage criteria ( $MC$ ) can be identified and mitigation test paths ( $MT$ ) can be created. A State Event Matrix ( $SE$ ) determines which failure types are possible in which behavioral states. This matrix and the test paths  $BT$  are then used in combination with failure scenario coverage criteria ( $CC$ ) to determine mitigation test requirements. The failure mitigation test paths ( $FMT$ ) are then created based on selecting an appropriate mitigation test ( $m \in MT$ ) and weaving it into the behavioral test according to the weaving rules associated with the selected mitigation test. These are then transformed into executable tests, executed, and validated. Note, that this process does not address corrective maintenance after failures are found, nor adaptive, or perfective maintenance or enhancements, all of which have the potential for selective regression testing. The following sections describe each of these steps in more detail. We generate test cases from the FSMWeb behavioral model (Section 3.2).

## 3.2 FSMWeb Approach

Functional testing for a web application follows the approach in [51]:

- Build a hierarchical model  $HFSM$ :
  - Partition the web application into clusters ( $Cs$ ).
  - Define Logical Web Pages ( $LWPs$ ) and Input-Action constraints for each.



- Build FSMs for clusters as a multi-level hierarchy.
- Build an Aggregate FSM (*AFSM*) to represent the top level of the application.
- Generate tests from the *HFSM*.
  - Generate paths through each FSM that meet the coverage criteria.
  - Aggregate paths form abstract tests.
  - Choose inputs along the paths to create executable tests.

The term *cluster* is used to refer to collections of software modules/web pages that implement a logical, user level function. The first step partitions the web application into clusters. At the highest level of abstraction, clusters represent functions that can be identified by users. At a lower level, clusters represent cohesive software modules/web pages that work together to implement a portion of a user level function.

Many web pages contain HTML forms, each of which can be connected to a different back-end software module. To facilitate testing for these modules, web pages are modeled as multiple *Logical Web Pages* (LWPs). A LWP is either a physical web page or the portion of a web page that accepts data from the user through an HTML form and then sends the data to a specific software module. FSMWeb is a functional model meant for black-box testing. Hence the web application can be written in any language appropriate for web applications (e.g. HTML, JavaScript, .. etc.). LWPs are abstracted from the presentation defined by the HTML and are described in terms of their sets of *inputs* and *actions*. All inputs in a LWP are considered atomic: data entered into a text field is considered to be only one user input symbol, regardless of how many characters are entered into the field. There

may be rules about the inputs. Some inputs may be required; others may be optional; users may be allowed to enter inputs in any order, or a specific order may be required. Table 3.1 shows the input constraints for both types while Table 3.2 shows how typical input types found in web applications are represented as constraints on (single) edges in an FSMWeb model.

Table 3.1: Constraints on Inputs

Input Choice	Order
Required ( <b>R</b> )	Sequence ( <b>S</b> )
Required Value ( <b>R</b> (parm))	Any ( <b>A</b> )
Optional ( <b>O</b> )	
Single Choice ( <b>C1</b> )	
Multiple Choice ( <b>Cn</b> )	

Table 3.2: FSMWeb Constraint of Typical Input Types

Input Type	FSMWeb Edge Annotation
Text Field Text Area Field	R (input name)
Optional Text Field Optional Text Area Field Optional Checkbox	O (input name)
Radio Box Drop Down Box (with $n$ options)	C1 (option 1, ..., option $n$ )
Optional Radio Box (with $n$ options)	O (C1 (option 1, ..., option $n$ ))
Set of Checkboxes Multi-Select Box (with $n$ options requiring 0 to $n$ selections)	O (Cn (option 1, ..., option $n$ )) A (option 1, ..., option $n$ )

This edge annotation via input-action constraints compresses an FSMWeb model because options for input selection and sequencing no longer need to be coded explicitly (which would inflate a traditional state-based model). Figure 3.1 shows how an FSM model that represents a selection with only three choices is reduced to two nodes and one transition in FSMWeb.

The lowest level cluster FSMs are generated with only LWPs and navigation between them. Input-action constraints annotate each edge [51]. Higher level FSMs represent FSMs from a lower level cluster by a single node and may contain LWP

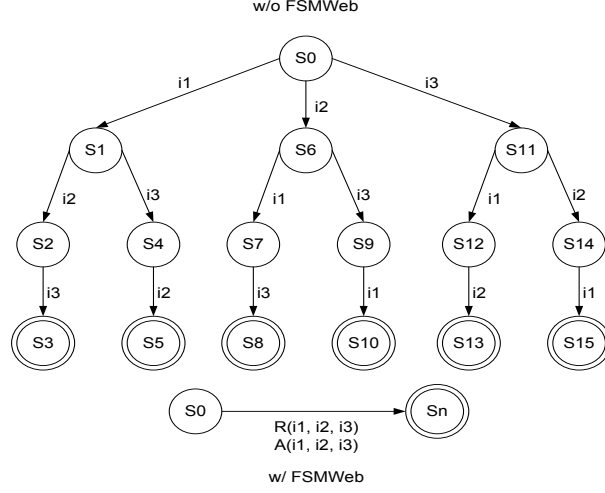


Figure 3.1: Three Optional Inputs, Any Order

nodes as well. Ultimately, a top-level Aggregate Finite State Machine (AFSM) is formed and represents a finite state model of the application at the highest level of abstraction.

Test sequences are generated during phase 2 of the FSMWeb method. A test sequence is a sequence of transitions through the application FSM and through each lower level FSM. FSMWeb’s test generation method first generates paths through each FSM based on some graph coverage criterion such as *edge coverage*. 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* [51].

This process results in a set of aggregate paths. We call them *abstract tests*. The final step of the test generation is selecting inputs to replace the input constraints for the transitions of the aggregate paths.

Input selection uses a technique [186] that builds two databases: a *synthetic database*, which consists of values that are consumed during testing, and an *application database*, which contains values previously inserted by the application being

tested. Values are saved into the application database during execution and saved into the synthetic database during testing. Details about the database creation and input selection can be found elsewhere [186].

Hence, an  $HFSM = \{FSM_i\}_{i=0}^n$  with a top level  $FSM_0 = AFSM$ . Each FSM has nodes that represent LWPs or clusters. Edges are internal or external to an FSM. External nodes span cluster boundaries. (They become internal at the next higher level.) External edges can either enter or leave a cluster FSM.

The FSM Tool parses HTML files and builds a FSMWeb model. The user can select coverage criteria such as node, edge, edge-pair, simple round trip and prime path coverage. The FSM Tool then generates test paths through each cluster that satisfy the selected criteria. For aggregation criteria, FSM Tool offers all-combinations, each choice and base choice coverage.

### 3.2.1 Example

The leftmost column of Figure 3.2 shows three FSMs and two levels of hierarchy<sup>1</sup>. This is an example of a behavioral model ( $BM$ ). Solid circles represent LWP nodes, the others are cluster nodes (i.e  $c_1$  and  $c_2$  in AFSM). It also shows  $FSM1$  and  $FSM2$  for  $c_1$  and  $c_2$  clusters. Table 3.3 shows paths through each FSM that achieve edge coverage. These test paths are aggregated to form abstract tests through the AFSM. As aggregation coverage criterion we use all combinations [51]. We illustrate this on  $t_{01} = n_1c_2n_2$ . Substituting  $t_{21}$  and  $t_{22}$  for  $c_2$  results in two paths:  $p_1 = n_1n_5n_7n_2$  and  $p_2 = n_1n_5n_6n_7n_2$ . Both paths consist of LWP nodes and do not have to be aggregated further.

Aggregation of  $t_{02}$  which visits cluster node  $c_1$  requires aggregation of one test

---

<sup>1</sup>For simplicity, we omitted input predicates. The right half shows changes to the model which are discussed in Section 4.

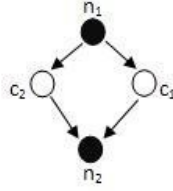
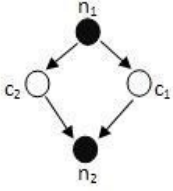

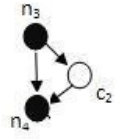
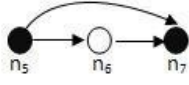
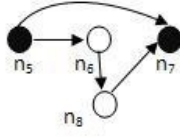
BM	BM'	Changes
AFSM 	AFSM 	
FSM1 	FSM1' 	Add Cluster Add Two Edges Delete Edge
FSM2 	FSM2' 	Add Node Add Two Edges Delete Edge

Figure 3.2: Behavioral Models  $BM$ ,  $BM'$

Table 3.3: Test Paths Through AFSM,FSM1,FSM2

FSM	Test paths
$AFSM$	$t_{01} = n_1 c_2 n_2, t_{02} = n_1 c_1 n_2$
$FSM_1$	$t_{11} = n_3 n_4 n_3$
$FSM_2$	$t_{21} = n_5 n_7, t_{22} = n_5 n_6 n_7$

path through  $FSM_1$ . This results in one path. There are 3 paths when test paths are fully aggregated. Table 3.4 shows these paths, including derivation rules used and test path lengths.

### 3.3 External Failures (F), State-Event matrix (SE)

We only need to test proper failure mitigation for those failure types that have mitigation requirements as stated in the software requirements description. Let

Table 3.4: Test Paths for  $BM$ 

Test	Test Paths $BT$	Derivation Rules	Full test path	Length
$t_{01} : n_1 c_2 n_2$				
$bt_1$	$n_1 t_{21} n_2$	$c_2 \rightarrow t_{21}$	$n_1, n_5, n_7, n_2$	4
$bt_2$	$n_1 t_{22} n_2$	$c_2 \rightarrow t_{22}$	$n_1, n_5, n_6, n_7, n_2$	5
$t_{02} = n_1 c_1 n_2$				
$bt_3$	$n_1 t_{11} n_2$	$c_1 \rightarrow t_{11}$	$n_1, n_3, n_4, n_3, n_2$	5

$F = \{f_1, \dots, f_k\}$  be the failure types, and  $S = \{s_1, \dots, s_n\}$  be the behavioral states. Failures may not be applicable in all behavioral states. For example, a failure type such as "expired session" is not applicable in the start state of a web application, since no session has been started yet. We express this in a State-Event matrix  $SE$  where element  $se_{ij}$  is given by:

$$se_{ij} = \begin{cases} 1, & \text{if failure type } j \text{ applies in node } s_i \in S \\ 0 & \text{otherwise.} \end{cases}$$

Using our example in section 3.2.1, for simplicity, we assume 4 failure types (see column 2 in Table 3.7 ).  $SE$  is defined as shown in Table 3.5, columns 2-8 and rows 2-5.

Table 3.5: State-Event Matrix  $SE$ 

Behavioral States ( $N$ )/ Failure Type ( $f$ )	$n_1$	$n_2$	$n_3$	$n_4$	$n_5$	$n_6$	$n_7$
1	1	0	0	1	1	0	1
2	0	0	1	0	1	1	0
3	0	1	0	1	1	1	1
4	0	1	0	0	0	0	1

### 3.4 Potential Failure Scenarios

Unlike Andrews *et al.* [70], we define mitigation test requirements based on three items:

- $i$  - the test  $bt_i$  in the test suite  $BT = \{bt_1, \dots, bt_l\}$   $1 \leq i \leq l$
- $p$  - the position in test  $bt_i$   $1 \leq p \leq \text{length}(bt_i)$
- $e$  - the type of failure  $1 \leq e \leq |F|$

Feasible failure scenarios are:

$$\{(i, p, e) | 1 \leq i \leq l, 1 \leq p \leq \text{Length}(bt_i), 1 \leq e \leq |F|, SE_{(\text{node}(p), e)} = 1\} \text{ where node } (p) \text{ is the index of the behavioral state in position } p \text{ of } bt_i$$

A mitigation test requirement  $(i, p, e)$  thus states in which test  $bt_i$  and where (position  $p$ ) in test  $bt_i$ , a particular failure of type  $e$  is to be applied to test its proper mitigation. We call these *potential failure scenarios*.

Note that we only apply a failure  $f_e$  once in a given test. Depending on the mitigation, applying a second failure later in the test may not even be possible, for example when the mitigation requires going to a safe state and stopping (the remainder of the behavioral test is discarded and no other failures are possible).

Table 3.6 shows the potential failure scenario matrix SP. Row 1 shows the index of  $bt$ . Row 2 of Table 3.6 shows the behavioral tests (from Table 3.4 in section 3.2.1). Row 3 identifies positions  $p$  in each test. Row 4 lists the sequence of nodes for each test. The remaining rows state a '1' when a failure is applicable in a state, a '0', if it is not.

Table 3.6: Potential Failure Scenarios  $SP$ 

i	1				2					3				
bt	$bt_1$				$bt_2$					$bt_3$				
Position ( $p$ )	1	2	3	4	1	2	3	4	5	1	2	3	4	5
F/N	$n_1$	$n_5$	$n_7$	$n_2$	$n_1$	$n_5$	$n_6$	$n_7$	$n_2$	$n_1$	$n_3$	$n_4$	$n_3$	$n_2$
$f_1$	1	1	1	0	1	1	0	1	0	1	0	1	0	0
$f_2$	0	1	0	0	0	1	1	0	0	0	1	0	1	0
$f_3$	0	1	1	1	0	1	1	1	1	0	0	1	0	1
$f_4$	0	0	1	1	0	0	0	1	1	0	0	0	0	1

### 3.5 Generate Test Requirements

We use the following coverage criteria from Andrews et al. [48] in this paper<sup>2</sup>.

**Criteria:** All tests, all unique nodes, all applicable failures. This requires that when unique nodes need to be covered they are selected from tests that have not been covered. The shaded '1' entries in Table 3.6 fulfill this criteria.

Coverage criteria are attractive since they allow for the systematic algorithmic generation of failure scenarios. The result of this step is the set of failure scenarios PE that constitute the failure mitigation test requirements.

### 3.6 Mitigation Requirements and Models

Common mitigation requirements include:

- Compensate ((Partial) Fix and Proceed): When a failure is raised, mitigation has to treat the failure and then return to the state where the failure occurred.
- Go to Fail-Safe State (Fix and Stop): When a failure cannot be mitigated and proceeding with a test case is not advisable due to adverse effects, the system

---

<sup>2</sup>This is a coverage criteria that performed well in a simulation study [69].



is forced to enter a fail-safe state (including shutting down).

- Rollforward (End Activity): This mitigation pattern will ship part of the remainder of the test.
- Rollbackward: Brings the system back to a previous state before the failure occurred.
- Retry: When a failure is detected an action is performed to resolve the failure and the activity that caused the problem is tried again.
- Internal compensate (no user action required). This can happen if a system switches to a backup activity or hardware that solves the problem.

Mitigation requirements can be expressed in the form of mitigation models. Figure 3.3 shows an example.

Each failure  $f_j$  is associated with a corresponding mitigation model  $MM_j$  where  $j = 1, \dots, k$ . The models are of the same type as the behavioral model BM. Graph-based [41], mitigation coverage criteria  $MC_j$  can be used to generate mitigation test paths  $MT_j = mt_{j_1}, \dots, mt_{j_{h_l}}$  for failure  $f_j$ .

Using the example of failure types in section 3.3, the corresponding mitigation requirements are summarized in Table 3.7 with the corresponding mitigation models.

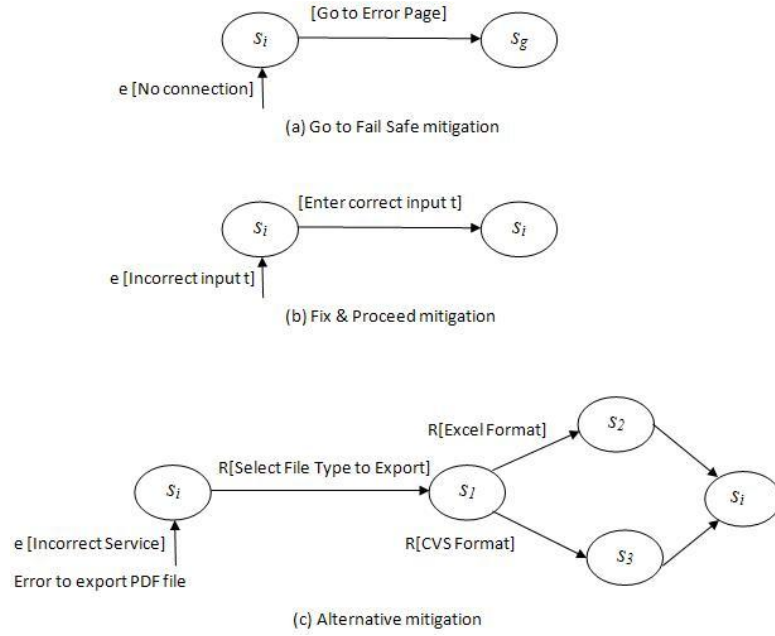


Figure 3.3: Mitigation Models.

Table 3.7: Mitigation Requirements

MM	Explanation	Model
MM1	Go to Fail Safe State: keep the system running even if there is no connectivity	see Figure 3.3-a, $MT_1 = \{mt_{11}\}$ where $mt_{11} = s_i, s_g$ and $s_g = LWP : errorpage$
MM2	End All: session expired, start over from the start node	$MT_2 = \phi$ and $s_b = n_1$ , where $s_b$ is the start node
MM3	Fix & proceed: parameter incompatibility such as data mismatch	see Figure 3.3-b, $MT_3 = \{mt_{31}\}$ where $mt_{31} = s_i, s_i$
MM4	Alternative: incorrect service	see Figure 3.3-c, $MT_4 = \{mt_{41}, mt_{42}\}$ where $mt_{41} = s_i, s_1, s_2, s_{i+1}$ and $mt_{42} = s_i, s_1, s_3, s_{i+1}$

### 3.7 Weaving Rules and Test Generation

Based on test requirements, PE, mitigation test paths are woven into the behavioral tests to create failure mitigation tests. Given a test requirement (i,p,e) failure  $f_e$  is applied at position p in the behavioral test  $bt_i$  and mitigation test  $mt_e$  is applied subject to weaving rules  $wr_e$ . Table 3.8 states weaving rules formally for each type of mitigation. Let  $t = \{s_1, \dots, s_b, \dots, node(p), \dots, s_f, \dots, s_k\}$ . Let  $s_g$  be a fail safe state.

Table 3.8: Mitigation Patterns and Weaving Rules

Mitigation pattern	Weaving Rule Name	WR#
Alternative $fmt = s_1 \dots node(p) \ mt \ node(p) \dots s_k$	Fix - option 1	1
Retry $fmt = s_1 \dots node(p) \ node(p)^r \dots s_k$	Rollback - option 3	2
Fix and Proceed $fmt = s_1 \dots node(p) \ mt \ node(p) \dots s_k$	Fix - option 1	3
End Activity $fmt = s_1 \dots node(p) \ mt \ s_f \dots s_k$	Rollforward - option 1	4
End All $fmt = s_1 \dots node(p) \ mt \ s_b$	Rollback - option 2	5
Rollback $fmt = s_1 \dots node(p) \ mt \ s_b \dots s_k$	Rollback - option 1	6
Ignore Internal compensate	No user action required	7
Go to fail-safe $fmt = s_1 \dots node(p) \ mt \ s_g$	Fix - option 2	8

Using the example in section 3.2.1, Table 3.9 shows the selected triplet and the  $fmts$  created based on them. The first column in Table 3.9 numbers each failure mitigation test ( $fmt_1 - fmt_{16}$ ). The second column lists each (i,p,e) triplet. The third column refers to the failure type whose mitigation is tested. The fourth column

states the node at position  $p$ . The fifth column identifies the behavioral test used in constructing  $f_{mt_i}$  ( $i = 1, \dots, 16$ ). The sixth column identifies which mitigation model is used as described in Table 3.7. The seventh column lists which  $mt_{ij}$  is used as described in Table 3.7. The seventh column shows the failure mitigation test. The last column states its length.

Table 3.9: Selected  $(i, p, e)$  Triplets and Resulting  $FMT$ .

#	Triplets	Failure	Node	BT used	$mt_{ij}$ used	FMT	Length
1	(1,1,1)	$f_1$	$n_1$	$bt_1$	$mt_{11}$	$n_1, s_g$	2
2	(1,2,1)	$f_1$	$n_5$	$bt_1$	$mt_{11}$	$n_1, n_5, s_g$	3
3	(1,3,1)	$f_1$	$n_7$	$bt_1$	$mt_{11}$	$n_1, n_5, n_7, s_g$	4
4	(3,3,1)	$f_1$	$n_4$	$bt_3$	$mt_{11}$	$n_1, n_3, n_4, s_g$	4
5	(1,2,2)	$f_2$	$n_5$	$bt_1$	$mt_{21}$	$n_1, n_5, n_1$	3
6	(2,3,2)	$f_2$	$n_6$	$bt_2$	$mt_{21}$	$n_1, n_5, n_6, n_1$	4
7	(3,2,2)	$f_2$	$n_3$	$bt_3$	$mt_{21}$	$n_1, n_3, n_1$	3
8	(1,2,3)	$f_3$	$n_5$	$bt_1$	$mt_{31}$	$n_1, n_5, n_5, n_7, n_2$	5
9	(1,3,3)	$f_3$	$n_7$	$bt_1$	$mt_{31}$	$n_1, n_5, n_7, n_7, n_2$	5
10	(1,4,3)	$f_3$	$n_2$	$bt_1$	$mt_{31}$	$n_1, n_5, n_7, n_2, n_2$	5
11	(2,3,3)	$f_3$	$n_6$	$bt_2$	$mt_{31}$	$n_1, n_5, n_6, n_6, n_7, n_2$	6
12	(3,3,3)	$f_3$	$n_4$	$bt_3$	$mt_{31}$	$n_1, n_3, n_4, n_4, n_3, n_2$	6
13	(1,3,4)	$f_4$	$n_7$	$bt_1$	$mt_{41}$	$n_1, n_5, n_7, s_1, s_2, n_2$	6
14	(1,3,4)	$f_4$	$n_7$	$bt_1$	$mt_{42}$	$n_1, n_5, n_7, s_1, s_3, n_2$	6
15	(1,4,4)	$f_4$	$n_2$	$bt_1$	$mt_{41}$	$n_1, n_5, n_7, n_2, s_1, s_2, n_2$	7
16	(1,4,4)	$f_4$	$n_2$	$bt_1$	$mt_{42}$	$n_1, n_5, n_7, n_2, s_1, s_3, n_2$	7

### 3.8 Generate Tests, Execute and Validate

The set of failure mitigation test paths now have to be transformed into executable tests. For the FSMWeb model, this means resolving the input predicates (*cf.* section 3.2). The sequences of inputs form the executable tests. These are now executed. The failure is injected at the proper position in the test execution. Execution monitoring is used to reveal mitigation defects.

## Chapter 4

### Approach Improvements

#### 4.1 Mortgage System Case Study: Fail-Safe Testing

##### 4.1.1 Case Study Research Questions

We used a case study to explore the following research questions:

1. RQ1: Applicability. So far, this approach has only been evaluated with respect to effectiveness and efficiency of generating test requirements via a GA using a simulation [44]. No case study has ever been done. Can it be applied to an actual system?
2. RQ2: Scalability. Much of the approach of generating FMT is manual. How feasible is this for a large system?
3. RQ3: Effectiveness. While simulation results show superior effectiveness to random search, will this approach be effective in a case study?
4. RQ4: Efficiency. How efficient is this approach or, to put it another way, how much effort is involved in the systematic testing of proper failure mitigation?

We explore these questions in the following sections.

### 4.1.2 Mortgage System

The mortgage system is an example of a critical web application as failures can be drastic: borrowers lose their home, the company loses its value, and employees lose their jobs. Not all system components are critical, e.g., components not related to the loan process. The system provides different services in each stage of the loan process. It includes the following functions:

1. Create the loan.
2. Acknowledge the loan based on the type of pricing.
3. Review the loan to make an approval decision.
4. Request legal documentation via external web services (document disclosure).
5. Keep and update accounting information for funding and selling the loan through selected warehouse banks and investors who buy loans.
6. Close the loan by shipping and tracking the loan's data with the investor.
7. Manage various data used in the system, e.g., adding/editing/deleting users or investors via an administration tool.
8. Provide utilities for loan processing, e.g., import/export loan data.

Software characteristics of the Mortgage System are shown in Table 4.1: It lists average cyclomatic complexity per class, maximum depth of inheritance, maximum class coupling, lines of code, number of files, size on disk, number of SQL database tables and number of logical web pages. We used the code metrics tool in Visual Studio for all numbers except the number of Logical Web Pages (LWP). The average complexity per class is  $170,476/2934 = 58$ . Size is measured for type file folder as

"size on disk". The system consists of 6,887 files in 1002 folders. The number of web pages is the number of logical web pages determined as per the FSMWeb modeling approach. Figure 4.1 shows a logical view of the mortgage system and its HTML links. The next step is to build the behavioral model.

Table 4.1: Software Characteristics of the Mortgage System [70]

Avg Cyclomatic Complexity per class	58
Max. Depth of Inheritance	9
Max. Class Coupling	1490
Lines of Code	257592
Number of Files	6887
Size in Gigabytes (on Disk)	1.48
SQL database Tables	204
Number of Logical Web Pages (LWP)	127

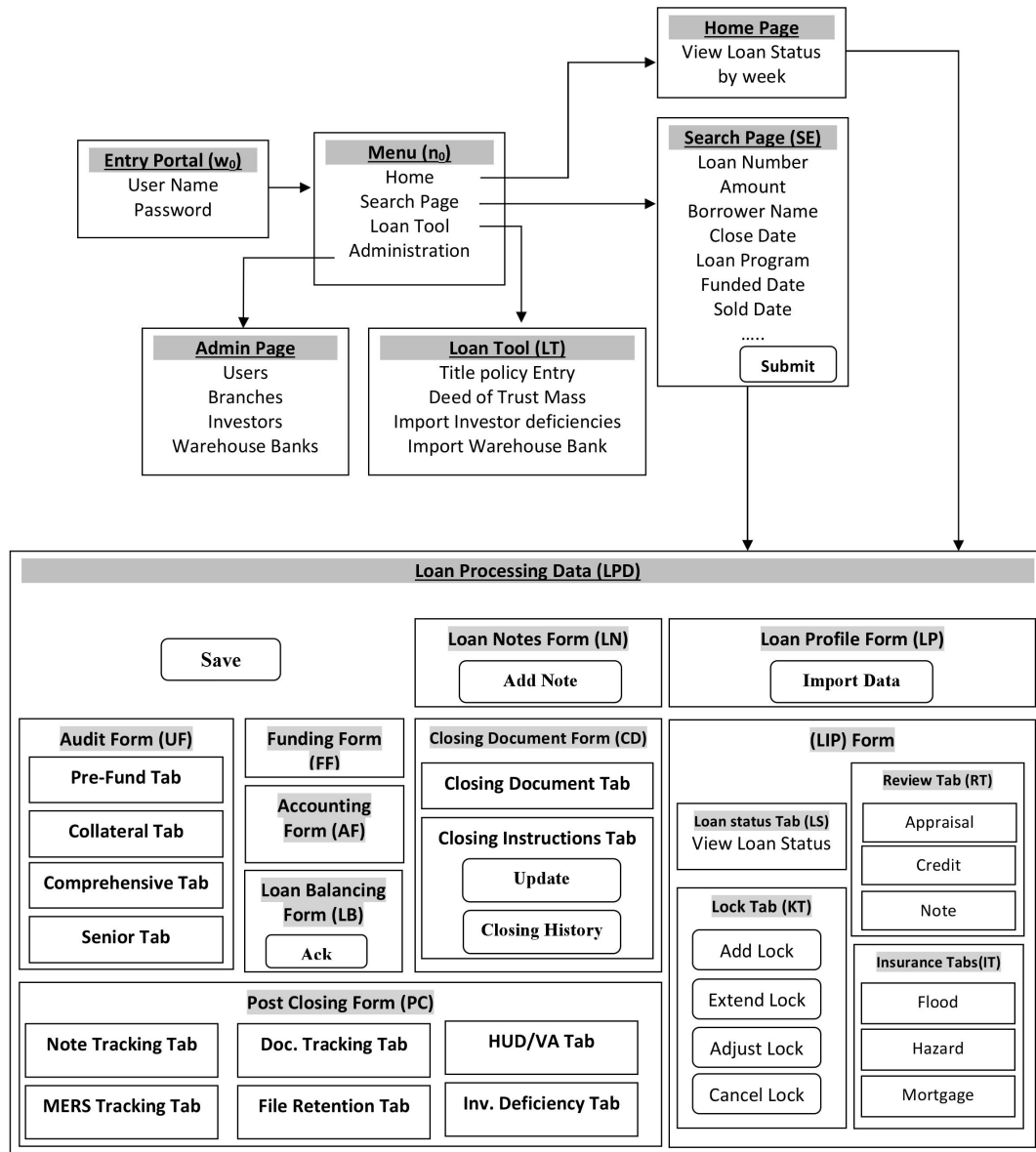


Figure 4.1: Mortgage System Logical View



### 4.1.3 FSMWeb Behavioral Model

#### 4.1.3.1 Partition FSMWeb Model

The highlighted items in Figure 4.1 identify the clusters for the FSMWeb model. Table 4.2 shows the partitions. The full model of this commercial web application consists of 127 LWPs, 22 clusters, and 224 transitions. Due to page limitations, we present only a key portion to illustrate the approach. The Closing Documents (CD) cluster is a lower level cluster in the LPD cluster (cf. Table 4.2). Figures 4.2-4.4 show the Aggregate FSM for the mortgage system, and the FSMs for Loan Processing Data and Closing Documents Service, respectively. For more detail see Boukhris [68]. Figure 4.4 shows the navigation among the logical web pages that represent the Closing Documents (CD) service. Table 4.3 shows the nodes and logical web pages related to CD.

What is interesting in this FSMWeb model is that the individual cluster FSMs are relatively small and compact, while an approach that models the whole application as one graph would be too large to handle manually.<sup>1</sup>

#### 4.1.3.2 Input Constraints for Logical Web Pages

Table 4.4 shows all input constraints for the Aggregate FSM (see Figure 4.2). The FSM input constraints for LPD of Figure 4.3 are shown in Table 4.5. Table 4.6 lists all FSM input constraints for the CD cluster as shown in Figure 4.4. In these input constraint tables, the leftmost column encodes each set of constraints ( $\Sigma$ ). The second column describes the action of the LWP, the third column identifies the constraints, the fourth column specifies transitions by their edges, and the fifth

---

<sup>1</sup>The full model has 127 nodes and 224 transitions

Table 4.2: Decomposition of Mortgage System into Partitions

AFSM	Cluster		Loan status
Entry Portal ( $w_0$ )			Any
Home Page			Any
Search Page (SP)	Simple Search / Advance Search		Any
Loan Processing Data (LPD)	Loan Notes Form (LN)		Any
	Loan Profile Form (LP)		Open
	LIP	Loan status tab (LS)	Any
		Lock Tab (KT)	Open-Lock
		Review Tab (RT)	Lock-Suspended
		Insurance Tab (IT)	Approve with condition
	Closing Document Tab (CD)		Lock-Suspended
	Funding Form (FF)		Approve with condition
	Accounting Form (AF)		Approve to close
	Audit Form (UF)		Funded - Sold
	Post Closing Form (PC)		Approve to close
	Loan Balance Form (LB)		Approve with condition
	Note (LN)		Any
	Title policy Entry (LT1)		Funded - Sold
Loan Tool (LT)	Deed of Trust Mass (LT2)	Funded - Sold	
	Import Investor deficiencies (LT3)	Funded - Sold	
	Import Warehouse Bank (LT4)	Funded - Sold	
Admin	Users ( $A_1$ )		NA
	Branches ( $A_2$ )		
	Investors ( $A_3$ )		
	Warehouse Banks ( $A_4$ )		

Table 4.3: Nodes for CD Form-FSM

Node	LWP	Explanation
$n_{p3}$	Selection tab menu	Select service from tab
DC	Documents to close	Request legal document, show, requested documents
CI	Closing Instructions	Request legal document, show, closing fees
SI	Show past instructions	Show the history of requested documents

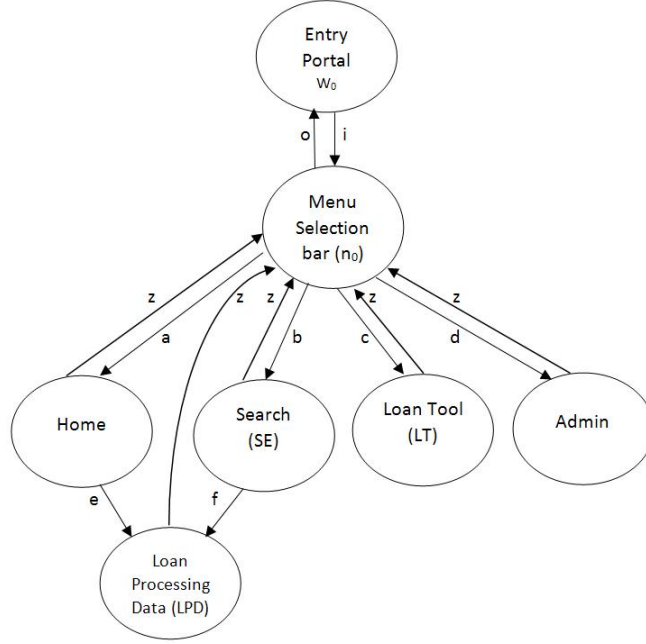


Figure 4.2: Aggregate FSMs with Partition and Top Level Navigation

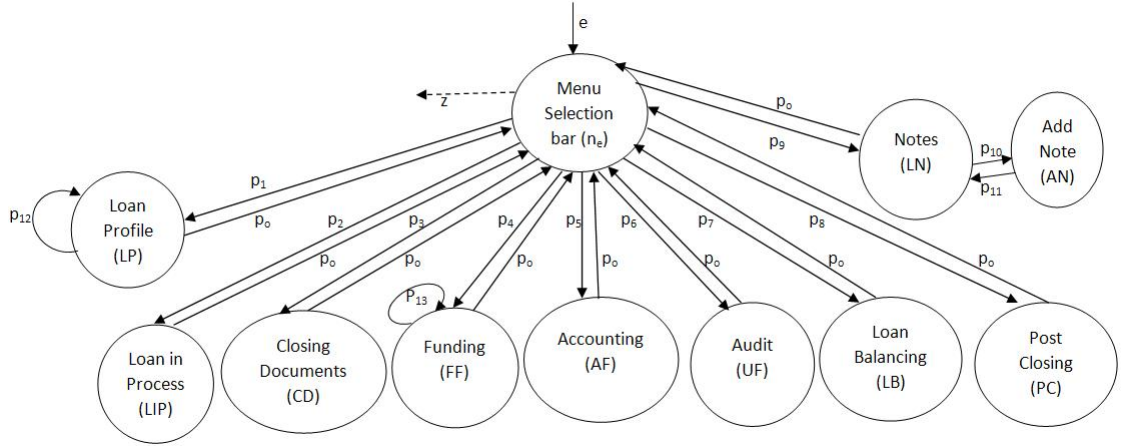


Figure 4.3: Loan Processing Data (LPD) Cluster

column ( $\Omega$ ) lists the next node or output. This information is used to provide a partial test oracle.

Table 4.4: Input Constraints of Aggregate FSM for Mortgage System

$\Sigma$	Actions	Constraints	Transition	$\Omega$
z	Dummy edge, reflect access to menu bar	$R(n_0, \text{Click})$	$(\text{Home}/\text{SE}/\text{LT}/$  $\text{LPD}/\text{Admin}, n_0)$	$n_0$
a	Access Home tab	$R(\text{tab}=\text{Home}, \text{Click}),$ $\text{Any}(\text{User Type}, \text{Loan Status})$	$(n_0, \text{Home})$	Home
b	Access Search tab	$R(\text{tab}=\text{SP}, \text{Click}), \text{Any}(\text{User Type}, \text{Loan Status})$	$(n_0, \text{SE})$	SE
c	Access Loan Tool tab	$R(\text{tab}=\text{LT}, \text{Click}),$  $R(\text{User Type in (CU, QC, AU, RU)},$ $\text{Loan Status in (Open, Lock, Approve to Close, Funded, Sold)})$	$(n_0, \text{LT})$	LT
d	Access Admin tab	$R(\text{tab}=\text{Admin}, \text{Click}),$  $R(\text{User Type}=\text{Admin})$	$(n_0, \text{Admin})$	Admin
e	Access to LPD	$R(A(\text{Borrower Name}, \text{Click})),$ $A(\text{User Type}, \text{Loan Status})$	$(\text{Home}, \text{LPD})$	LPD
f	Access to LPD	$O(R(\text{Loan Number}, \text{Click}),$ $R(\text{Search Result match Specific Loan}, \text{Click}))$	$(\text{SE}, \text{LPD})$	LPD
i	Login the sys- tem	$R(\text{User Name}, \text{Password})$	$(w_0, n_0)$	$n_0$
o	Log out the sys- tem	$R(\text{Logout}, \text{Click})$	$(w_0, n_0)$	$w_0$

Table 4.5: Input Constraints for LPD Cluster

$\Sigma$	Actions	Constraints	Transition	$\Omega$
$p_o$	Dummy edge, reflect Back to $n_e$	$R(n_e, \text{Click})$	$(LP/LIP/CD/FF/AF/UF/LB/PC/LN, n_e)$	$n_e$
$p_1$	Access LP tab	$R(\text{tab}=LP, \text{Click})$ Editing: $R(\text{User type}=LO, \text{Loan status}=\text{Open})$ Viewing: $A(\text{User type}, \text{Loan status})$	$(n_e, LIP)$	LIP
$p_2$	Access LIP tab	$R(\text{tab}=LIP, \text{Click}), A(\text{User type}, \text{Loan Status})$	$(n_e, LIP)$	LIP
$p_3$	Access CD tab	$R(\text{tab}=CD, \text{Click})$ Editing: $R(\text{User type}=SU, \text{Loan Status in Approve to Close, Approve with Condition})$ Viewing: $A(\text{User type}, \text{Loan status})$	$(n_e, CD)$	CD
$p_4$	Access FF tab	$R(\text{tab}=FF, \text{Click})$ Editing: $R(\text{User type}=AU, \text{Loan Status in Approve to Close, Funded, Sold})$ Viewing: $A(\text{User type}, \text{Loan status})$	$(n_e, FF)$	FF
$p_5$	Access AF tab	$R(\text{tab}=AF, \text{Click})$ Editing: $R(\text{User type}=AU, \text{Loan Status in Approve to Close, Funded, Sold})$ Viewing: $A(\text{User type}, \text{Loan status})$	$(n_e, AF)$	AF
$p_6$	Access UF tab	$R(\text{tab}=UF, \text{Click})$ Editing: $R(\text{User type}=QC, A(\text{Loan Status}))$ Viewing: $A(\text{User type}, \text{Loan status})$	$(n_e, UF)$	UF
$p_7$	Access LB tab	$R(\text{tab}=LB, \text{Click})$ Editing: $R(\text{User type in QC, LO, Loan Status in Approve to Close, Approve with Condition})$ Viewing: $A(\text{User type}, \text{Loan status})$	$(n_e, LB)$	LB
$p_8$	Access PC tab	$R(\text{tab}=PC, \text{Click})$ Editing: $R(\text{User type}=CU, \text{Loan Status in (Funded, Sold)})$ Viewing: $A(\text{User type}, \text{Loan status})$	$(n_e, PC)$	PC
$p_9$	$A(\text{User type}, \text{Loan Status})$	$R(\text{tab}=LN, \text{Click})$  Editing/Viewing: $A(\text{User type}, \text{Loan Status})$	$(n_e, LN)$	LN
$p_{10}$	Add note	$R(\text{Add}, \text{Click})$	$(LN, AN)$	AN
$p_{11}$	Back to Notes	$R(O(\text{Save}, \text{Cancel}), \text{Click})$	$(AN, LN)$	LN
$p_{12}$	Update Loan profile by calling external web service	$R((\text{import button}, \text{Click}), \text{User type in Admin, LO, Loan Status}=\text{Open})$	$(LP, LP)$	LP
$p_{13}$	Revert Funding	$R((\text{Revert button}, \text{Click}), \text{User type}=AU, \text{Loan Status}=\text{Funded})$	$(FF, FF)$	FF
$z$	Back to $n_0$	$R(n_0, \text{Click})$	$(n_e, n_0)$	$n_0$

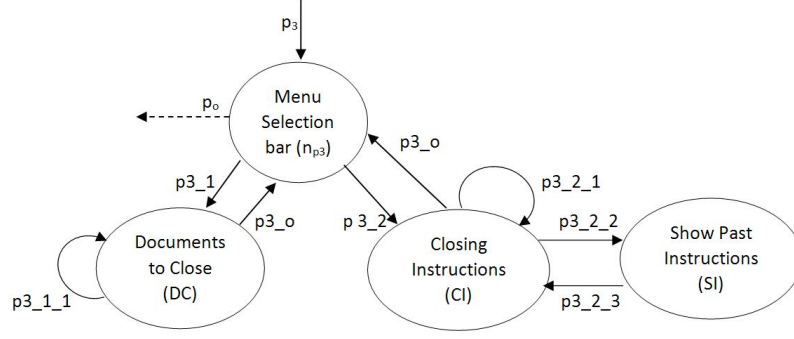


Figure 4.4: FSM For Closing Documents (CD) Service

As an example, the second row in Table 4.6 is a transition from  $n_{p3}$  (menu selection bar) to access the Documents to Close page (DC) tab. The code in column 1 ( $p3\_1$ ) is used as an abbreviation for the full constraint in column 3. The output DC (in the column marked  $\Omega$ ) is the target state of the transition.

Input constraints can vary widely in size, depending on how much data needs to be input. The Closing Document cluster only has a limited number of inputs since the loan information has already been entered and most of the inputs merely relate to different closing actions.

#### 4.1.3.3 Generate Test Paths through Clusters

For the Closing Documents (CD) service, we apply transition coverage to the FSM, generating test sequences to cover each transition. Test sequences for the CD FSM are shown in Table 4.7. The first column indicates the test path through CD. The second column indicates the constraint sequence (using the abbreviations defined in the  $\Sigma$  column in Table 4.6). The third column elaborates the input constraints for this test path.

Test paths for the Aggregate FSM are shown in Table 4.8 and for the LPD cluster in Table 4.9. Given that the cluster FSMs are fairly compact, the paths are

Table 4.6: Input Constraint for Closing Documents (CD) FSM

$\Sigma$	Actions	Constraints	Transition	$\Omega$
p3_1	Access CD	R((tab= CD, Click), User type=SU, Loan Status in Approve to close, Approve with condition )	$(n_{p3}, DC)$	DC
p3_2	Access CI	R((tab= CI, Click), User type=SU, Loan Status in Approve to close, Approve with condition )	$(n_{p3}, CI)$	CI
p3_1_1	Calling external web service	R((Sync from IDS, Click), User type=SU, Loan Status in Approve to close, Approve with condition )	(DC, DC)	DC
p3_2_1	Calling external web service	R((Sync from IDS, Click), User type=SU, Loan Status in Approve to close, Approve with condition )	(CI, CI)	CI
p3_2_2	Access SI	R((Show Past Instructions, Click), User type=SU, Loan Status in Approve to close, Approve with condition )	(CI, SI)	SI
p3_2_3	Back to CI	R(Close, Click)	(SI, CI)	CI
p3_o	Back to $n_{p3}$	R( $n_{p3}$ , Click)	$(DC/CI, n_{p3})$	$n_{p3}$
$p_o$	Back to $n_e$	R( $n_e$ , Click)	$(n_{p3}, n_e)$	$n_e$

Table 4.7: Test Sequence for Closing Documents (CD) FSM

Test Path	Constraint Sequence	Constraint
$(n_{p3}, DC, n_{p3})$	p3_1, p3_o	R((tab= CD, Click), User type=SU, Loan Status in Approve to close, Approve with condition )
$(n_{p3}, CI, SI, CI, n_{p3})$	p3_2, p3_2_2, p3_2_3, p3_o	$\mathcal{Q}(R((tab= CI, Click), R((Show Past Instructions, Click))), User type=SU, Loan Status in Approve to close, Approve with condition )$
$(n_{p3}, CI, CI, n_{p3})$	p3_2, p3_2_1, p3_2_2, p3_o	$\mathcal{R}(R((tab= CI, Click), User type=SU, Loan Status in Approve to close, Approve with condition )$

fairly short and limited in size. Obviously, the coverage criterion influences required test paths and their length. For example, prime path coverage would have required more tests. What is important to notice is that there is no problem with scalability related to test paths required for the CD cluster. This confirms scalability results in [50].

Table 4.8: Test Paths of Aggregate FSM for the Mortgage System

Test #	Path
$T_0$	$w_0, n_0, \text{Home}, n_0, w_0$
$T_1$	$w_0, n_0, \text{Home}, \text{LPD}, n_0, w_0$
$T_2$	$w_0, n_0, \text{SE}, n_0, w_0$
$T_3$	$w_0, n_0, \text{SE}, \text{LPD}, n_0, w_0$
$T_4$	$w_0, n_0, \text{LT}, n_0, w_0$
$T_5$	$w_0, n_0, \text{Admin}, n_0, w_0$

Table 4.9: Test Paths of FSM for LPD Cluster

Test #	Path
$T_{LPD1}$	$n_e, \text{LP}, \text{LP}, n_e$
$T_{LPD2}$	$n_e, \text{LIP}, n_e$
$T_{LPD3}$	$n_e, \text{CD}, n_e$
$T_{LPD4}$	$n_e, \text{FF}, \text{FF}, n_e$
$T_{LPD5}$	$n_e, \text{AF}, n_e$
$T_{LPD6}$	$n_e, \text{UF}, n_e$
$T_{LPD7}$	$n_e, \text{LB}, n_e$
$T_{LPD8}$	$n_e, \text{PC}, n_e$
$T_{LPD9}$	$n_e, \text{LN}, \text{AN}, \text{LN}, n_e$

#### 4.1.3.4 Aggregate Paths to Generate Abstract Tests

Table 4.10 summarizes the number of tests for the whole mortgage system as well as for the CD Cluster. The first row lists the number of tests through individual cluster FSMs. Next, we list their length as well as the number of aggregated abstract test paths and their length. We used the FSM Tool to generate the paths



for the whole system. Inputs were selected according to Ran et al. [186]. Generation, execution, and validation of the behavioral test suite took approximately two staff months of effort (assuming 160 hours per month). We need on average 12.5 min/webpage to generate, execute, and validate a test. This is a reasonable effort for a system of this size. It also demonstrates further the scalability of FSMWeb which was already shown with different case studies [50].

Table 4.10: Statistics of Tests Size

	Mortgage System	CD Cluster
Number of tests through FSMs	106	3
Size of test suite (Number of nodes)	127	12
Number of aggregated tests	266	12
Size of aggregated test suite	3998	169

As a more detailed example, we will perform the substitution for  $T_1$  and  $T_3$  paths from Table 4.8. This means substituting test sequences in Table 4.9 for LPD in order to resolve all test paths for CD. This substitution results in a total of 12 abstract test paths as shown in Tables 4.11 and 4.12. (VC,VC) is a path through the Home Cluster. The paths starting with SP and ending in SR are paths through the SE cluster. Due to space limitations, their FSMs are not shown.

Table 4.11: Paths Generated by Substitution of  $T_1$

Clusters	Test No.	Test Path
Home-LPD-CD	$T_{CD1}$	$w_0, n_0, VC, VC, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$
Home-LPD-CD	$T_{CD2}$	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, CI, n_{p3}, n_e, n_0, w_0$
Home-LPD-CD	$T_{CD3}$	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$

#### 4.1.4 Failure Applicability and Mitigation Requirements

We will detail the cluster Closing Documents(CD). Table 4.13 lists mitigations for all failure types and gives an example of each. Corresponding mitigation re-

Table 4.12: Paths Generated by Substitution of  $T_3$ 

Clusters	Test No.	Test Path
SE-LPD-CD	$T_{CD4}$	$w_0, n_0, \text{SP}, \text{SR}, \text{EE}, \text{SR}, n_e, n_{p3}, \text{DC}, \text{DC}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD5}$	$w_0, n_0, \text{SP}, \text{SP}, \text{AS}, \text{SR}, n_e, n_{p3}, \text{DC}, \text{DC}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD6}$	$w_0, n_0, \text{SP}, \text{AS}, \text{SP}, \text{AS}, \text{SR}, n_e, n_{p3}, \text{DC}, \text{DC}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD7}$	$w_0, n_0, \text{SP}, \text{SR}, \text{EE}, \text{SR}, n_e, n_{p3}, \text{CI}, \text{SI}, \text{CI}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD8}$	$w_0, n_0, \text{SP}, \text{SP}, \text{AS}, \text{SR}, n_e, n_{p3}, \text{CI}, \text{SI}, \text{CI}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD9}$	$w_0, n_0, \text{SP}, \text{AS}, \text{SP}, \text{AS}, \text{SR}, n_e, n_{p3}, \text{CI}, \text{SI}, \text{CI}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD10}$	$w_0, n_0, \text{SP}, \text{SR}, \text{EE}, \text{SR}, n_e, n_{p3}, \text{CI}, \text{CI}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD11}$	$w_0, n_0, \text{SP}, \text{SP}, \text{AS}, \text{SR}, n_e, n_{p3}, \text{CI}, \text{CI}, n_{p3}, n_e, n_0, w_0$
SE-LPD-CD	$T_{CD12}$	$w_0, n_0, \text{SP}, \text{AS}, \text{SP}, \text{AS}, \text{SR}, n_e, n_{p3}, \text{CI}, \text{CI}, n_{p3}, n_e, n_0, w_0$

quirements are summarized in Table 4.14 which also specifies the corresponding mitigation models and associated weaving rules for each failure type. The last column in the table refers to the weaving rule number defined in Table 3.8. Table 4.15 shows the State-Event matrix for the CD cluster. It indicates that not all failure types are applicable in all states. For example,  $f_1$  (no network connection) is applicable in all states.  $f_2$  (session is expired) is not applicable for the entry portal web page (node  $w_0$ ). Similarly,  $f_6$  (user switches back and forth in the browser) can occur in all states except  $w_0$ . In the DC state, all failure types except  $f_{10}$  can occur (DC does not export data). What is interesting is that the mitigation requirements cover all types of mitigations and all weaving rules. Further, most mitigation models are fairly compact, indicating that failure mitigation tests involve only a few extra actions. This shows that only a relatively small proportion of failure scenarios are feasible, reducing potential test requirements.

From Table 4.10, we know that the CD cluster has a test suite of length 169. Given the 10 failure types, the total search space including infeasible positions is  $169 \times 10 = 1690$ . Because not all failures are applicable in all behavioral states, the feasible search space is 638, which is about 38% of all pairs. By contrast, the Mortgage system overall has a test suite length of 3998 with a total search space of

39980 (including infeasible pairs). The feasible search space is 13034 which is about 32% of all pairs.

Table 4.13: Failure Types in Cluster Closing Documents (CD)

Failure Type	Mitigation	Example
$f_1$ : unavailability	Go to Fail Safe State	No network connection
$f_2$ : time out	End ALL	Session expired
$f_3$ : Parameter incompatibility	Fix & proceed	Input error (Integer vs. string)
$f_4$ : response error	Rollback	Database server response error
$f_5$ : Misunderstood behavior	End Activity	Access tab needs specific user role.
$f_6$ : Work flow inconsistency	Ignore	Back and forth user browser navigation
$f_7$ : incorrect order	Fix & proceed	Required loan process step
$f_8$ : Browser incompatibility	Retry	Java Script for viewing does not work correctly
$f_9$ : Interface change	Roll Back	External service changes the mapping
$f_{10}$ : incorrect service	Alternative	Incorrect service to export data grid into a file

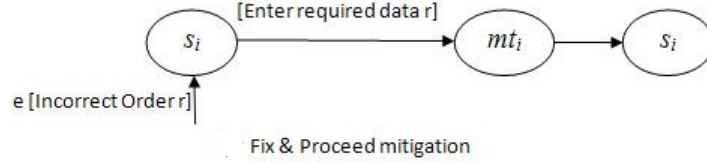


Figure 4.5: Mitigation Model  $MM_7$ .

#### 4.1.5 Test Requirements and their Effectiveness

To illustrate our approach, we use the GA on the CD cluster using the abstract tests determined in Table 4.11 and 4.12. Table 4.16 shows the behavioral test

Table 4.14: Mitigation Requirement for CD Cluster

MM	Explanation	Model	WR#
MM1	Go to Fail Safe State: keep the system running even if there is no connectivity	see Figure 3.3-a, $MT_1 = \{mt_{11}\}$ where $mt_{11} = (s_i, s_g)$ and $s_g = LWP : errorpage$	8
MM2	End All: session expired, so start over from the start node	$MT_2 = \phi$ and $s_b = w_0$ , where $s_b$ is the start node	5
MM3	Fix & proceed: parameter incompatibility such as data mismatch	see Figure 3.3-b, $MT_3 = \{mt_{31}\}$ where $mt_{31} = (s_i, s_i)$	3
MM4	Rollback: database server error	$MT_4 = \phi$ where $s_b = DC$ , and $s_b$ is DC state where trying to save data	6
MM5	End Activity: misunderstood behavior such as try to access CD cluster without having SU user role	$MT_5 = \phi$ where $s_f = n_{p3}$ , and $s_f$ is a menu selection bar of CD cluster	4
MM6	Ignore: work flow inconsistency such as using browser navigation incorrectly	Internal compensate	7
MM7	Fix & proceed: incorrect order	see Figure 4.5, $MT_7 = \{mt_{71}\}$ where $mt_{71} = (s_i, mt_i, s_i)$	3
MM8	Retry: Java Script error	$MT_8 = \phi$ where $node(p)^r$	2
MM9	Rollback: Interface change	$MT_9 = \phi$ where $s_b = n_{p3}$	6
MM10	Alternative: incorrect service	see Figure 3.3-c, $MT_{10} = \{mt_1, mt_2\}$ where $mt_1 = (s_i, n_1, n_2, s_i)$ and $mt_2 = s_i, n_1, n_3, s_i$	1

Table 4.15: State-Event Matrix for CD Cluster

Behavioral States/ Failure Type ( $f$ )	$w_0$	$n_0$	VC	$n_e$	$n_{p3}$	SP	SR	AS	EE	DC	CI	SI
1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	1	1	1	1	1	1	1
3	1	0	0	0	0	1	0	0	0	1	0	0
4	0	0	0	0	0	1	0	1	0	1	0	0
5	0	0	0	0	0	0	0	0	0	1	1	1
6	0	1	1	1	1	1	1	1	1	1	1	1
7	0	0	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	0	0	0	0	0	1	1	0
10	0	0	0	0	0	0	1	0	0	0	0	1

paths  $BT$  for them. It identifies the clusters the test covers, the test identifier, the sequence of nodes and the length of each test path. The last row shows the length of the concatenated test paths.

Table 4.16: Abstract Test Paths (CD Cluster)

Clusters	Test ID	Test Path	Length
Home-LPD-CD	$T_{CD1}$	$w_0, n_0, VC, VC, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$	12
Home-LPD-CD	$T_{CD2}$	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, CI, n_{p3}, n_e, n_0, w_0$	12
Home-LPD-CD	$T_{CD3}$	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$	13
SE-LPD-CD	$T_{CD4}$	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$	14
SE-LPD-CD	$T_{CD5}$	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$	14
SE-LPD-CD	$T_{CD6}$	$w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$	15
SE-LPD-CD	$T_{CD7}$	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$	15
SE-LPD-CD	$T_{CD8}$	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$	15
SE-LPD-CD	$T_{CD9}$	$w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$	16
SE-LPD-CD	$T_{CD10}$	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, CI, CI, n_{p3}, n_e, n_0, w_0$	14
SE-LPD-CD	$T_{CD11}$	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, CI, CI, n_{p3}, n_e, n_0, w_0$	14
SE-LPD-CD	$T_{CD12}$	$w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3}, CI, CI, n_{p3}, n_e, n_0, w_0$	15
Length of $ I $ :			<b>169</b>

Table 4.17 shows the failure mitigation test paths (FMT) for the first generation. The first column identifies the test. The second column shows the test requirements (p,e) used to construct the test. The next two columns state failure type and node at position  $p$ . The fifth column identifies the mitigation model as described in Table

4.14. The sixth and eighth columns state the resulting fail-safe mitigation test  $f_{mt}$  and which behavioral test  $bt$  was used to create it. Notice that  $f_{10}$  has two mitigation test paths for MM10 as explained in Table 4.14 and hence two failure mitigation tests for the pair (105,10).

Table 4.17: Initial FMT for First Generation

#	(p,e)	Failure	Node	MM used	FMT	Length	BT used	Length
1	(46,1)	$f_1$	DC	MM1	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, DC, S_g$	10	$T_{CD4}$	14
2	(19,2)	$f_2$	CI	MM2	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, w_0, n_0, VC, VC, n_e, n_{p3}, CI, CI, n_{p3}, n_e, n_0, w_0$	19	$T_{CD2}$	12
3	(40,3)	$f_3$	SP	MM3	$w_0, n_0, SP, SP, SR, EE, SR, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$	15	$T_{CD4}$	14
4	(56,4)	$f_4$	AS	MM4	$w_0, n_0, SP, SP, AS, AS, SR, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$	15	$T_{CD5}$	14
5	(32,5)	$f_5$	SI	MM5	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, SI, n_{p3}$	9	$T_{CD3}$	13
6	(41,6)	$f_6$	SR	MM6	$T_{CD4}$	14	$T_{CD4}$	14
7	(61,7)	$f_7$	DC	MM7	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, DC, mt_i, DC, DC, n_{p3}, n_e, n_0, w_0$	16	$T_{CD5}$	14
8	(75,8)	$f_8$	DC	MM8	$w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3}, DC, DC, DC, n_{p3}, n_e, n_0, w_0$	16	$T_{CD6}$	15
9	(8,9)	$f_9$	DC	MM9	$w_0, n_0, VC, VC, n_e, n_{p3}, DC, DC, n_{p3}$	9	$T_{CD1}$	12
10	(105,10)	$f_{10}$	SI	MM10	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, CI, SI, n_1, n_2, SI, CI, n_{p3}, n_e, n_0, w_0$	18	$T_{CD8}$	15
10	(105,10)	$f_{10}$	SI	MM10	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, CI, SI, n_1, n_3, SI, CI, n_{p3}, n_e, n_0, w_0$	18	$T_{CD8}$	15
11	(1,1)	$f_1$	$w_0$	MM1	$w_0, S_g$	2	$T_{CD1}$	12
12	(2,1)	$f_1$	$n_0$	MM2	$w_0, n_0, S_g$	3	$T_{CD1}$	12
13	(3,1)	$f_1$	VC	MM3	$w_0, n_0, VC, S_g$	4	$T_{CD1}$	12
14	(5,1)	$f_1$	$n_e$	MM4	$w_0, n_0, VC, VC, n_e, S_g$	6	$T_{CD1}$	12
15	(6,1)	$f_1$	$n_{p3}$	MM5	$w_0, n_0, VC, VC, n_e, n_{p3}, S_g$	7	$T_{CD1}$	12
16	(42,1)	$f_1$	EE	MM6	$w_0, n_0, SP, SR, EE, S_g$	6	$T_{CD4}$	14

Next, we explore the effectiveness of the GA for this case study. We assumed a mitigation defect rate of 5% and seeded the CD subsystem with three defects, similar to what was used in [44] as the lower defect density. Since the number of ones in the state-event matrix (see Table 4.15) is 50;  $50 \times 5\% = 2.5$ . The number of seeded defects is rounded up to 3.

Table 4.18 shows how many pairs and number of generations were needed to find all mitigation defects. The GA generated 352 (p,e) pairs to find 3 mitigation defects. There are a total of 12 generations.

According to Table 4.18, the GA finds the first defect in the first generation and it takes seven generations to find the next defect, but it takes only four more generations to find the third defect.

Table 4.18: Effectiveness of GA

# of Generation	# of Pairs	Defect Found %
initial population	16	33%
2-7	149	33%
8-11	146	67%
12	41	100%
Total pairs	352	

Table 4.19 shows the FMTs for the last generation based on 41 test requirements. Table 4.20 shows the *fmts* that detected the 3 mitigation defects. Pair (7,1) finds the defect for  $f_1$  (unavailability of network) in state DC using the behavioral test case  $T_{CD}=(w_0,n_0,VC,VC,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0)$ . The mitigation of  $f_1$  is to keep the system running even when there is no connectivity by going to the Fail Safe state  $s_g$ . This is an error page describing the defect and asking to contact system administration. The mitigation model is *MM1* as shown in Table 4.14. The failure mitigation test path (***fmt***) is given by:  $(w_0,n_0,VC,VC,n_e,n_{p3},DC,s_g)$ . Similarly, the mitigation defect for  $f_2$  is found by using the pair (32,2) (state SI in test path  $T_{CD3}$ ) and constructing the *fmt* using the weaving rule "End All" and starting over. Lastly, the mitigation defect for  $f_3$  is found using pair (60,3) (state DC in test path  $T_{CD5}$ ). The failure mitigation test is constructed by repeating the edge that showed the failure and then proceeding. In our case, it is the edge  $(DC,n_{p3})$ .

In Boukhris [68], we apply the GA to the whole system. We seed the same percentage of defects. This results in 25 defects. Table 4.21 shows the total number of generations needed and the cumulative number of pairs generated to expose all mitigation defects. The GA needs 42 generations to find all defects. For the whole

Table 4.19: FMT for Last Generation

#	(p,e)	Failure	Node	MM used	FMT	Length	BT used	Length
1	(33,3)	$f_3$	SI	MM3	$T_{CD3}$	13	$T_{CD3}$	13
2	(60,3)	$f_3$	DC	MM3	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, DC, DC,$ $DC, n_{p3}, n_e, n_0, w_0$	15	$T_{CD5}$	14
3	(75,4)	$f_4$	DC	MM4	$w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3},$ $DC, DC, DC, n_{p3}, n_e, n_0, w_0$	16	$T_{CD6}$	15
4	(165,4)	$f_4$	CI	MM4	$T_{CD12}$	15	$T_{CD12}$	15
5	(89,5)	$f_5$	CI	MM5	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, CI, n_{p3}$	10	$T_{CD7}$	15
6	(133,5)	$f_5$	$n_e$	MM5	$T_{CD10}$	14	$T_{CD10}$	14
7	(56,6)	$f_6$	AS	MM6	$T_{CD5}$	14	$T_{CD5}$	14
8	(135,6)	$f_6$	CI	MM6	$T_{CD10}$	14	$T_{CD10}$	14
9	(7,6)	$f_6$	DC	MM6	$T_{CD1}$	12	$T_{CD1}$	12
10	(105,6)	$f_6$	SI	MM6	$T_{CD8}$	15	$T_{CD8}$	15
11	(46,7)	$f_7$	DC	MM7	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, DC,$ $mti, DC, n_{p3}, n_e, n_0, w_0$	15	$T_{CD4}$	14
12	(90,7)	$f_7$	SI	MM7	$T_{CD7}$	15	$T_{CD7}$	15
13	(120,8)	$f_8$	CI	MM8	$w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3},$ $CI, CI, SI, CI, n_{p3}, n_e, n_0, w_0$	17	$T_{CD9}$	16
14	(19,9)	$f_9$	CI	MM9	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, n_{p3}$	8	$T_{CD2}$	12
15	(129,9)	$f_9$	SP	MM9	$T_{CD10}$	14	$T_{CD10}$	14
16	(148,9)	$f_9$	$n_{p3}$	MM9	$T_{CD11}$	14	$T_{CD11}$	14
17	(81,10)	$f_{10}$	$w_0$	MM10	$T_{CD7}$	15	$T_{CD7}$	15
18	(32,10)	$f_{10}$	SI	MM10	$w_0, n_0, VC, VC, n_e, n_{p3}, CI,$ $SI, n_1, n_2, CI, n_{p3}, n_e, n_0, w_0$	15	$T_{CD3}$	13
19	(32,10)	$f_{10}$	SI	MM10	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, SI,$ $n_1, n_3, CI, n_{p3}, n_e, n_0, w_0$	15	$T_{CD3}$	13
20	(159,1)	$f_1$	SP	MM1	$w_0, n_0, SP, AS, SP, s_g$	6	$T_{CD12}$	15
21	(47,1)	$f_1$	DC	MM1	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, DC, DC, s_g$	11	$T_{CD4}$	14
22	(56,1)	$f_1$	AS	MM1	$w_0, n_0, SP, SP, AS, s_g$	6	$T_{CD5}$	14
23	(107,1)	$f_1$	$n_{p3}$	MM1	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3},$ $CI, SI, CI, n_{p3}, s_g$	13	$T_{CD8}$	15
24	(15,1)	$f_1$	VC	MM1	$w_0, n_0, VC, s_g$	4	$T_{CD2}$	12
25	(7,1)	$f_1$	DC	MM1	$w_0, n_0, VC, VC, n_e, n_{p3}, DC, s_g$	8	$T_{CD1}$	12
26	(5,1)	$f_1$	$n_e$	MM1	$w_0, n_0, VC, VC, n_e, s_g$	6	$T_{CD1}$	12
27	(38,1)	$f_1$	$w_0$	MM1	$w_0, s_g$	2	$T_{CD4}$	14
28	(85,1)	$f_1$	EE	MM1	$w_0, n_0, SP, SR, EE, s_g$	6	$T_{CD7}$	15
29	(130,1)	$f_1$	SR	MM1	$w_0, n_0, SP, SR, s_g$	5	$T_{CD10}$	14
30	(119,1)	$f_1$	$n_{p3}$	MM1	$w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3}, s_g$	10	$T_{CD9}$	16
31	(153,1)	$f_1$	$n_0$	MM1	$w_0, n_0, SP, SP, AS, SR, n_e,$ $n_{p3}, CI, CI, n_{p3}, n_e, n_0, s_g$	14	$T_{CD11}$	14
32	(136,2)	$f_1$	CI	MM2	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, CI, CI, s_g$	11	$T_{CD10}$	14
33	(158,2)	$f_1$	AS	MM2	$w_0, n_0, SP, AS, s_g$	5	$T_{CD12}$	16
34	(6,2)	$f_1$	$n_{p3}$	MM2	$w_0, n_0, VC, VC, n_e, n_{p3}, s_g$	7	$T_{CD1}$	12
35	(7,2)	$f_2$	DC	MM2	$w_0, n_0, VC, VC, n_e, n_{p3}, DC, w_0, n_0,$ $VC, VC, n_e, n_{p3}, DC, DC, n_{p3}, n_e, n_0, w_0$	19	$T_{CD1}$	12
36	(32,2)	$f_2$	SI	MM2	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, SI, w_0, n_0, VC,$ $VC, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$	21	$T_{CD3}$	13
37	(157,2)	$f_2$	SP	MM2	$w_0, n_0, SP, w_0, n_0, SP, AS, SP, AS, SR, n_e, n_{p3},$ $CI, CI, n_{p3}, n_e, n_0, w_0$	18	$T_{CD12}$	15
38	(101,2)	$f_2$	SR	MM2	$w_0, n_0, SP, SP, AS, SR, w_0, n_0, SP, SP, AS,$ $SR, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$	21	$T_{CD8}$	15
39	(131,2)	$f_2$	EE	MM2	$w_0, n_0, SP, SR, EE, w_0, n_0, SP, SR, EE, SR,$ $n_e, n_{p3}, CI, CI, n_{p3}, n_e, n_0, w_0$	19	$T_{CD10}$	14
40	(152,2)	$f_2$	$n_e$	MM2	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, CI, CI, n_{p3},$ $n_e, w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, CI, CI, n_{p3},$ $n_e, n_0, w_0$	26	$T_{CD11}$	14
41	(94,2)	$f_2$	$n_0$	MM2	$w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, CI, SI, CI, n_{p3},$ $n_e, n_0, w_0, n_0, SP, SR, EE, SR, n_e, n_{p3}, CI,$ $SI, CI, n_{p3}, n_e, n_0, w_0$	29	$T_{CD7}$	15



system, the GA generated 8276 test requirements. The corresponding *FMT* has a length of 99312 nodes. As it did for the Closing Documents (DC) subsystem, the GA finds all mitigation defects.

Table 4.20:  $fmt_i$  that Found Defects

FMT	Failure	State	BT used	GA Pairs	MM used	Explanation
$fmt_1$	$f_1$	DC	$T_{CD1}$	(7,1)	MM1	$w_0, n_0, VC, VC, n_e, n_{p3}, DC, S_g$
$fmt_2$	$f_2$	SI	$T_{CD3}$	(32,2)	MM2	$w_0, n_0, VC, VC, n_e, n_{p3}, CI, SI, w_0, n_0, VC,$ $VC, n_e, n_{p3}, CI, SI, CI, n_{p3}, n_e, n_0, w_0$
$fmt_3$	$f_3$	DC	$T_{CD5}$	(60,3)	MM3	$w_0, n_0, SP, SP, AS, SR, n_e, n_{p3}, DC, DC, DC,$ $n_{p3}, n_e, n_0, w_0$

Table 4.21: Test Requirements and Mitigation Defects Found for Mortgage Lending System

# of Generation	# of Pairs	Defect Found %
initial population	170	0%
2-4	530	0%
5-11	1340	4%
12-16	1141	8%
17-18	583	16%
19	324	27%
20-22	1056	33%
23-24	751	42%
25	391	46%
26-27	817	53%
28	418	58%
29	430	67%
30-31	895	72%
32	457	75%
33-35	1385	77%
36	469	80%
37	472	85%
38-40	1428	93%
41	484	95%
42	485	100%

In summary, the test requirements generated by GA were successful in finding all mitigation defects for both the CD subsystem and the whole mortgage system. The size of the FMT is an order of magnitude larger than the behavioral test BT. Given that the weaving approach reuses a lot of the original BT, this helps to make

the generation of FMT more efficient. Due to the compact nature of the mitigation models, added mitigations are short, so most of the original BT can be reused. This can be seen, for example by comparing length of FMT (e.g. column 7 in Table 4.19) with length of BT (e.g. column 9 in Table 4.19)

#### 4.1.6 Comparison of Effort: GA vs. Exhaustive Search

At several points, we claimed that exhaustive search, i.e., converting all feasible pairs to executable tests, executing, and validating them is prohibitive. To investigate this claim, we measured the time it took to translate a set of test requirements into executable tests, executing and validating the results. We then computed average effort per node in the test path. We computed the length of the failure mitigation test suite for the exhaustive search for both the CD subsystem and the whole system and multiplied with the average effort per node to arrive at an effort estimate for exhaustive search.

Table 4.22 shows the results. For the CD subsystem, the test requirements using GA requires 352 pairs, the total length of all failure mitigation tests is 3565. Estimated average test effort is about six work days (note a work day=8 hours) while using exhaustive search requires more than 11 work days of testing. However, for the whole system, the differences are much more drastic: more than 155 work days for GA vs. about 525 work days for exhaustive search. This is more than three times as long. Given that both find all mitigation defects, the choice is obvious.

Table 4.23 shows the number of nodes, transitions, the total number of behavioral test paths  $BT$ , and their length. Note that the effort estimates reported in Table 4.22 refer to mitigation testing only and do not include testing primary functionality.

Table 4.22: Effort Comparison Between GA vs. Exhaustive Search

Approach	Test ments (# of pairs)	Require- ments (# of <i>length(FMT)</i> )	Time Es- timation (min)	Total Hours	Work Days
<b>The Subsystem (CD)</b>					
GA	352	3565	2674	45	5.6
Exhaustive search	638	7195	5396	90	11.25
<b>The Whole system</b>					
GA	8276	99312	74484	1241	155.23
Exhaustive search	27986	335920	251940	4199	524.88

Table 4.23: The Size of *BT* for CD vs. Mortgage System

	# of nodes	# of transitions	size of BT	Length (I)
CD subsystem	12	9	12	169
Mortgage System	127	224	266	3998

Given the large length of the behavioral test suite for the mortgage system, the search space is quite large (almost 40,000 entries). Note however that the SE Matrix only is based on the number of web pages (127) and the number of the failure types (10). We constructed it separately for the LWPs in each FSM, taking advantage of the clustering to keep each matrix as compact as possible.

#### 4.1.7 Findings

Even for the relatively large mortgage system, the behavioral model is still compact, primarily due to the FSMWeb’s clustering and input compression approach. Hence scalability even for a large web application is evident. We also took advantage of the clusters, using them to construct the SE matrix as a series of smaller matrices, one for each FSM’s LWPs. The number of behavioral tests is reasonable as is the time it took to test (about two person months).

The external failure mitigation requirements for the mortgage system include all the failure types defined in Section 3.3, demonstrating their applicability. Similarly, all eight weaving rules are needed. What is interesting is that the mitigation models

are very small.

Both the CD subsystem and the mortgage system as a whole have an applicability level of only a little over 30%, which is helpful in that it reduces possible mitigation testing requirements and thus mitigation test effort.

The weaving approach in conjunction with the fact that the mitigation actions are usually small, allow a large proportion of BT to be reused (see for example Table 4.19). This speaks to the efficiency of our approach.

Failure mitigation testing is an order of magnitude more expensive (1241 hours) than testing regular behavior (320 hours). This should come as no surprise, as industry practice has long confirmed that testing invalid and problem situations consumes more of a test team’s time than testing regular operation [162]. Fortunately, the GA approach presented here is less expensive than exhaustively testing all possible failure scenarios while being effective in finding all mitigation defects. We also investigated with the software development team how long they tested the mortgage system using their prior approach. The (rough) estimate was about six staff months. This is lower than what is needed for our approach but lacks the systematic approach to testing external failures and carries no assurance that all mitigations for external failures were properly tested. In summary, the case study showed that systematic testing for proper mitigation of external failures is both practically feasible and effective even for a relatively large web application.

#### 4.1.8 Threats to Validity

The threats are related to the choice of the parameters used in the GA. We address each in turn: the weights  $w_r$  and  $w_s$  are based on tuning experiments performed in Andrews *et al.* [44]. We follow the recommendations based on their

simulation results although they use a somewhat different fitness function.

Their tuning relies on the use of published mitigation defect rates (around 20%) (i.e Sawaelpong *et al.* [197]). They also experiment with a much lower defect rate (5%). This supports our use of a common defect rate of 20%, contrasting it with a low one (5%) as well. While it may be possible to tune these weights for higher efficiency, this would expose to potential overfitting. We are hence willing to accept that the GA is not always optimally tuned.

As for the choice of mutation rate and crossover, we use values that have been suggested in the literature. We used a crossover rate of CR=0.50 and a mutation rate of MR=0.30. Typical rates based on De Jong’s simulations are between 0.5-0.6 [73, 89]. As for the mutation rate, theoretical work reports a rule of thumb of  $1/N$  ( $N$  is the number of genes, in our case  $N = 2$ ) [56]. By contrast, typical mutation rates in the literature are around 0.15 [73]. The mutation rate we chose is a compromise between the two.

Like Patton et al. [181], we aim to provide the GA with a semi-ideal starting position. It has been known that good initial populations can reduce the number of individuals and generations during the search and increase the chance of finding a good solution [93]. Thus, it is preferable to seed the initial population with a possible or partial solution to the problem. For us, this means using defect potential to determine the initial population. We evaluate how good the initial population is by comparing an initial population selected via defect potential against multiple runs of randomly selected initial populations (10 runs). The results clearly show that using defect potential to generate test requirements outperforms random selection of the initial population.

Multiple runs are possible when the use of a GA is explored with a simulator as in [181][60]. However, when actual test cases need to be generated, executed and

validated to determine a test requirement’s fitness, this GA evaluation cost becomes prohibitive for multiple runs. We accept a local rather than global optimum as long as the mitigation defects are found. For quantitative results on evaluation, cost see Section 4.1.6 in our case study. Note also that the global minimum in terms of the number of test requirements is equal to the number of mitigation defects that exist. Additionally, Cantú-Paz and Goldberg [79] explore whether multiple runs can reach solutions of higher quality or reach acceptable solutions faster. Their results suggest that with a fixed evaluation budget a single run reaches a better solution than multiple independent runs.

As with any case study, generalizability is limited. We cannot guarantee that a future case study performs the same way. Similarly, seeding of faults for effectiveness evaluation in Section 4.1.5 may not exactly represent actual failure mitigation defects. However, this is an accepted practice in many studies [119]. We did show through this case study that the approach can be applied to a relatively large web application from the financial sector.

## 4.2 Regression Testing Approach

In Boukhris [68], an approach for regression testing was provided that used GA even when changes are small. As we discovered GA is not always advisable [69]. We, therefore, change the approach to use coverage criteria. We also noticed that the RT requirements were inefficient as they required to retest reusable test requirements. We modified the approach to remove reusable test requirements. We also did a significant case study and compared efficiency improvements to not having them. Additionally, we modified the testing process.

### 4.2.1 Process

Based on the changes to the artifact used in the test generation approach in Subsection 3.1, we classify tests as retestable, reusable, or obsolete. Only the steps from that point on the need to be repeated. Figure 4.6 shows the regression testing process and references the sections that describe the changes.

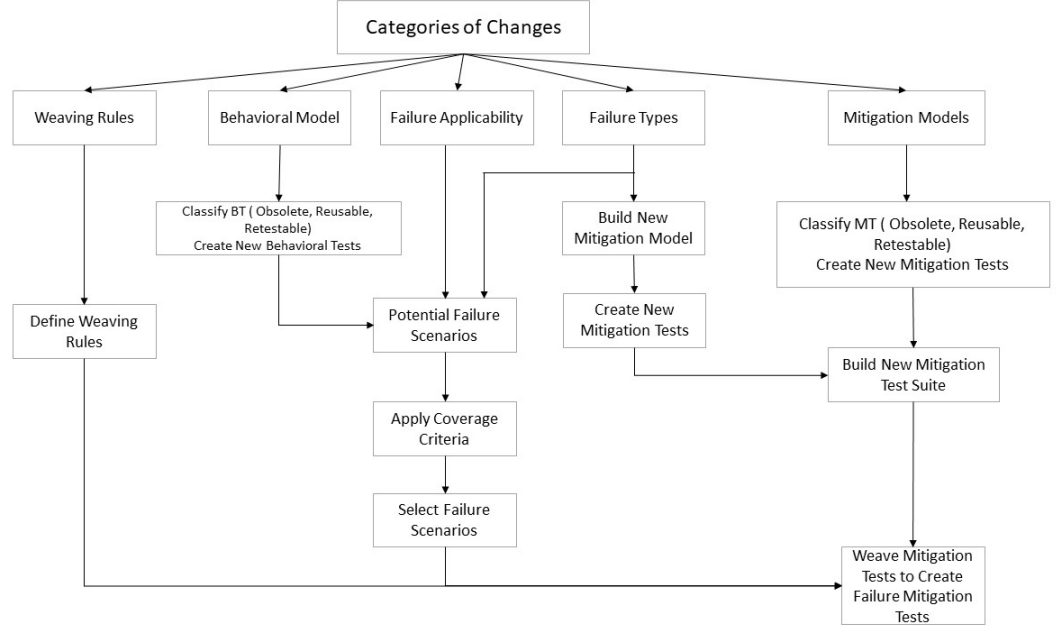


Figure 4.6: Regression Testing Process

The process steps to build a regression test suite outlined in Figure 4.6 is based on the types of changes to various artifacts:

1. Changes in the behavioral model:

We classify the behavioral tests as obsolete, retestable, or reusable using the classification rules in Andrews et al. [49]. We determine if any parts of the new behavioral model have not been tested and generate new tests for them. We construct the new behavioral test suite. If there are no changes to  $F$ , we build a new state-event matrix and potential failure scenarios, then apply coverage

criteria to select failure scenarios. In case of added failure types, we add the new failure types to the new state-event matrix and the new potential failure scenarios. If mitigation models are not changed, we use the mitigation tests from the existing mitigation models for each failure type. Then we generate the new failure mitigation tests. In case the mitigation models are changed, we classify the mitigation tests as obsolete, retestable, or reusable. We generate new mitigation tests if necessary. Next, we weave the new mitigation test suite to generate new failure mitigation tests. In case of changes to the weaving rules, we apply the new weaving rules to the mitigation tests if there are no changes to mitigation models; otherwise, we use the new weaving rules with the new mitigation test suite to generate new failure mitigation tests.

## 2. Changes in the state-event matrix:

If there are no changes to the behavioral model or the failure types, the changes to the state-event matrix require building a new set of potential failure scenarios. We select failure scenarios based on coverage criteria. If mitigation models are not changed, we then weave the mitigation tests to create the new failure mitigation tests.

If there are changes to the behavioral model as well, we follow the same steps that are described in step one and generate the new behavioral test suite. We add any new nodes as new columns in the new state-event matrix. We build the new potential failure scenarios.

If there are added failure types, we simply extend the state-event matrix to include the new failure types and build the new potential failure scenarios based on all changes. Next, we apply coverage criteria to select failure scenarios and generate new failure mitigation tests. If there are any changes to the



mitigation models or weaving rules, we also apply step 4 and 5 below.

3. Changes in failure types:

Adding new failure types requires building a new state-event matrix by adding a new row to the matrix for each new failure type. Then, we construct potential failure scenarios and select failure scenarios for the new failure types only. We build new mitigation models for the new failure types, and create new mitigation tests.

If there are no changes to the behavioral model and weaving rules, we generate the new failure mitigation tests by weaving the new mitigation tests into behavioral tests. Otherwise, we use the new behavioral test suite from step 1 and generate new failure mitigation tests. In case of changes to the mitigation models or weaving rules, we follow steps 4 and 5 below. If failure types are deleted, we simply delete all the associated mitigation models, weaving rules and failure mitigation tests for the deleted failure types.

4. Changes in mitigation models:

We only need to consider changes to individual mitigation models since all other changes to various artifacts are already covered. We classify the mitigation tests for the changed mitigation model into obsolete, retestable, or reusable. We generate new mitigation tests for edges that are not covered in the modified mitigation model. This process applies to each mitigation model that has changed. Next, we use the new mitigation tests to build the new failure mitigation tests. If there are added failure types, we include the new mitigation tests derived from the new mitigation models that have been built for the new failure types with new failure scenarios to build failure mitigation on tests. In case of changes to weaving rules, we apply the new weaving rules

to the new mitigation test suite to generate new failure mitigation tests.

5. Changes in weaving rules:

We reweave all mitigation tests using the new weaving rules to generate the new failure mitigation tests. All changes to other artifacts have been dealt with in prior steps.

6. Execute the failure mitigation test suite.

In determining how to execute the test suite, considerations like avoiding cross-contamination of tests need to be taken into account. This may require bringing the System Under Test (SUT) into a known state before executing a test. While test execution environments like Selenium [23] can be used to execute tests, injecting external failures (testing their mitigation is the point of our approach) would still have to be covered separately. Another consideration in making the test suite executable is how test methods are written. The use of assertions has been shown to correlate with test effectiveness [238]. Our approach allows making tests executable in whatever way the tester deems appropriate. Thus assertions may be used in this step, but do not have to be used.

7. Validate Results.

Tests can pass or fail. When they fail, it can be due to a defect in the system under test (SUT), a defect in the model, or a defect in the test(s) derived from the model. Validation determines which is the case.

## 4.2.2 Detailed Approach

Now, we describe the regression test generation in detail and illustrate each step with a running example. We define changes to the models in the example of Section 3.1 as we formalize the algorithms in each step to illustrate how they work. Figure 3.2 shows three FSMs and two levels of hierarchy. This is our behavioral model. It also shows  $FSM1$  and  $FSM2$  before and after changes resulting in  $FSM1'$  and  $FSM2'$ . Table 3.3 and Table 3.4 in section 3.2.1 show paths through each FSM that achieve edge coverage, including derivation rules and the test path lengths. In section 3.3, the failure applicability matrix  $SE$  is defined as shown in Table 3.5. In section 3.4, Table 3.6 shows the potential failure scenarios. In section 3.7, Table 3.7 shows the corresponding mitigation models and associated weaving rules for each failure type and Table 3.9 shows the selected failure scenarios and resulting failure mitigation tests.

### 4.2.2.1 Changes to the Behavioral Model (BM)

The right side of Figure 3.2 shows both the original and the changed behavioral model. In  $FSM1$ , edge  $(n_4, n_3)$  was deleted and cluster node coverage criteria and edges  $(n_3, c_2), (c_2, n_4)$  were added. In  $FSM2$ , edge  $(n_6, n_7)$  was deleted. LWP node  $n_8$  and edges  $(n_6, n_8), (n_8, n_7)$  were added.

#### 4.2.2.1.1 Classify $BT$ into obsolete, reusable and retestable behavioral tests

The test classification rules are as follows:

- **Obsolete Tests Paths ( $BT_o$ )**

Andrews et al. [49] define a set of rules for defining obsolete test paths based on the type of change: node deletion affects all paths that include the deleted

node, rendering it obsolete. Node modifications change the type of node from LWP to a cluster node or vice versa. This type of modification affects the paths as the node needs to be replaced by another node or a sequence of nodes. Thus, both node deletions and node modifications render test paths obsolete that tour these nodes: let the set of behavioral test paths be  $BT = \{bt_1, \dots, bt_l\}$ . Let  $N_o = \{n \mid n \in N; n \text{ is deleted or modified} \}$  where  $N$  is the set of behavioral nodes, then the set of obsolete test paths due to node changes is given by

$O_N = \{bt_i \mid \exists n \in N_o : bt_i \text{ visits } n\}$ , where  $bt_i$  is a behavioral test path. For our example,  $N_o = \phi$ , and  $O_N = \phi$ .

Edge deletion makes any test paths that tour the edge obsolete. Any edge modification that requires changes in the inputs, guards, actions, outputs, and messages associated with it, will also make test paths obsolete that visit the modified edge.

Let  $E_o = \{e \mid e \in E; e \text{ is deleted or modified}\}$  where  $E$  is the set of behavioral edges, then the set of obsolete test paths due to edge changes is given by

$O_E = \{bt_i \mid t_a \text{ tours } e \in E_o \}$ . For our example,  $E_o = \{(n_4, n_3), (n_6, n_7)\}$ , and  $O_E = \{bt_2, bt_3\}$ . Thus, the set of obsolete behavioral test paths is given by:  $BT_O = O_N \cup O_E$ . Hence  $BT_o = \phi \cup O_E = \{bt_2, bt_3\}$ .

- **Retestable Test Paths ( $BT_r$ )**

In Andrews et al. [49], retestable tests are defined as those that are still valid and test portions of the application and visit part of the FSMWeb model that are affected by the changes. This can be determined in different ways. For example, Andrews et al. [49] consider any node  $n$  that is one edge away from a modified or deleted node as impacted by the change, except for the source

and sink nodes of the AFSM. Using this definition:

$$N_{r_{node}} = \{n \mid \exists e : (n, \hat{n}) \text{ or } (\hat{n}, n); \hat{n} \in N_o; n \neq n_{source}; n \neq n_{sink}\}.$$

Since in our example  $N_o = \phi$ ,  $N_{r_{node}} = \phi$  as well. When edges are changed, the set of retestable edges depends on the type of change. When edges are deleted or modified, we assume that the starting and ending nodes of the changed edges are potentially impacted and hence nonobsolete tests that visit these nodes are retestable (except for the source and sink nodes of the model):

$$N_{r_{edm}} = \{n \mid \hat{e} \in E_o; \hat{e} = (n_i, n) \text{ or } \hat{e} = (n, n_i); n \neq n_{source}; n \neq n_{sink}\}. \text{ In the example } N_{r_{edm}} = \{n_3, n_4, n_6, n_7\}.$$

Similarly, when we add new edges, existing nodes at which these new edges start or end are considered potentially affected by the modification and hence non-obsolete tests that visit these nodes are retestable (Except for the source and sink nodes of the model):

Let  $E$  be the set of edges in  $BM$ . Let  $E'$  is the set of added edges in  $BM'$ . Then  $E' \setminus E$  is the set of added edges.

$$N_{r_{ea}} = \{n \mid n \in N_j : \hat{e} = (n, n_i) \text{ or } \hat{e} = (n_i, n); \hat{e} \in E' \setminus E, n_i \in N'; n \neq n_{source}; n \neq n_{sink}\}$$

In the example,  $E' \setminus E = \{(n_3, c_2), (c_2, n_4), (n_6, n_8), (n_8, n_7)\} \implies N_{r_{ea}} = \{n_3, n_4, n_6, n_7\}$ . This happens to be the same as  $N_{r_{edm}}$ .

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 paths is

$$BT_r = \{bt_i \mid bt_i \text{ visits } n \in N_r, bt_i \in BT\} \setminus BT_o$$

In the example,  $N_r = \{n_3, n_4, n_6, n_7\} \implies BT_r = \{bt_1, bt_2, bt_3\} \setminus \{bt_2, bt_3\} = \{bt_1\}$

In our example, the only non-obsolete test path  $bt_1$  is also retestable.

- **Reusable Test Paths ( $BT_u$ )**

Those tests are neither obsolete nor retestable.  $BT_u = BT \setminus (BT_o \cup BT_r)$ . The example has no reusable test paths  $BT_u = \phi$ .

- **New Test Paths ( $BT'$ )**

New test cases need to be designed whenever current test cases do not meet coverage requirements for  $BM'$ . This happens when edges or nodes are added. Obsolete test cases can also cause gaps in coverage that need to be addressed with new tests. In the example, the following edges in the modified model  $BM'$ :  $\{(n_3, c_2), (c_2, n_4), (n_6, n_8), (n_8, n_7)\}$  are not covered. Thus, new test paths are generated:  $bt'_1 = \{n_1, n_3, n_4, n_2\}$ ,  $bt'_2 = \{n_1, n_3, n_5, n_7, n_4, n_2\}$ ,  $bt'_3 = \{n_1, n_3, n_5, n_6, n_8, n_7, n_4, n_2\}$ , and  $bt'_4 = \{n_1, n_5, n_6, n_8, n_7, n_2\}$ . As a result, the new test paths are  $BT' = \{bt'_1, bt'_2, bt'_3, bt'_4\}$ .

The new test suite will be as follows:  $BT'' = BT_r \cup BT'$ . In the example, Table 4.24 shows the classification of behavioral test paths  $BT$  and the new test path  $BT'$  for the modified model. So, the new test suite will be as follows:

$$BT'' = BT_r \cup BT' = \{bt_1, bt'_1, bt'_2, bt'_3, bt'_4\}.$$

Table 4.24: Classification of  $BT$  After the Changes

$BT$	Path	Classification
$bt_1$	$n_1, n_5, n_7, n_2$	Retestable
$bt_2$	$n_1, n_5, n_6, n_7, n_2$	Obsolete
$bt_3$	$n_1, n_3, n_4, n_3, n_2$	Obsolete

$BT'$	Path	Classification
$bt'_1$	$n_1, n_3, n_4, n_2$	New
$bt'_2$	$n_1, n_3, n_5, n_7, n_4, n_2$	New
$bt'_3$	$n_1, n_3, n_5, n_6, n_8, n_7, n_4, n_2$	New
$bt'_4$	$n_1, n_5, n_6, n_8, n_7, n_2$	New

#### 4.2.2.1.2 Build New $SE'$ matrix

Any change in the BM such as adding, modifying, or deleting a node requires to rebuild the state event(SE) matrix. When adding a new node, we add a new column to SE, and similarly, when deleting a state, we delete a column from SE.

The new state-event matrix  $SE'$  is shown in Table 4.25. It is the same as the original  $SE$  except for the added column for the new states  $n_8$ , and  $c_2$ .

Table 4.25: New State-Event Matrix  $SE'$

Behavioral States/ Failure Type ( $f$ )	$n_1$	$n_2$	$n_3$	$n_4$	$n_5$	$n_6$	$n_7$	$n_8$
1	1	0	0	1	1	0	1	0
2	0	0	1	0	1	1	0	1
3	0	1	0	1	1	1	1	0
4	0	1	0	0	0	0	1	0

#### 4.2.2.1.3 Build New Potential Failure Scenarios $SP'$ and new Failure Mitigation Tests $FMT''$

The new potential failure scenarios consider the new tests  $BT'$  and the retestable  $BT_r$  ( $BT'' = BT' \cup BT_r$ ). This requires constructing a new potential failure scenario matrix  $SP'$ . The new potential failure scenario matrix  $SP'$  uses  $BT''$  instead of  $BT$ . We use  $SP'$  to select new triplets  $PE'$ . We assume that any changes in mitigation models have been dealt with and the new mitigation test suite  $MT'$  has been built or that  $MT' = MT$  if no changes occurred. We generate failure mitigation tests  $FMT'$  with  $PE'$ . Row 1 in Table 4.24 identifies the 4 test paths in  $(BT'')$ . Row 2 shows the new  $(BT')$  and retestable test path and row 3 shows their node sequences.

The shaded entries in Table 4.26 mark triplets that fulfill our coverage criterion using [48]. However, some of the failure scenario triplets do not have to be retested (i.e. the requirements are *reusable*), because generating failure mitigation tests from

Table 4.26: New Potential Failure Scenarios  $SP'$ 

$BT''$	1				2						3								4						5				
	$bt'_1$				$bt'_2$						$bt'_3$								$bt_4$						$bt_1$				
	1	2	3	4	1	2	3	4	5	6	1	2	3	4	5	6	7	8	1	2	3	4	5	6	1	2	3	4	
p	$n_1$	$n_3$	$n_4$	$n_2$	$n_1$	$n_3$	$n_5$	$n_7$	$n_4$	$n_2$	$n_1$	$n_3$	$n_5$	$n_6$	$n_8$	$n_7$	$n_4$	$n_2$	$n_1$	$n_5$	$n_6$	$n_8$	$n_7$	$n_2$	$n_1$	$n_5$	$n_7$	$n_2$	
$f_1$	1	0	1	0	1	0	1	1	1	0	1	0	1	0	0	1	1	0	1	1	0	1	0	1	0	1	1	1	0
$f_2$	0	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	0	0	0	1	0	0	
$f_3$	0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	1	1	0	1	1	0	1	1	1	
$f_4$	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	0	0	1	

them would result in failure mitigation tests that were already in the original failure mitigation test suite FMT (i.e. have been run).

For example, generating a new  $f_{mt}$  from (1,1,1) would result in  $f_{mt_1}$  in Table 3.9. The reusable triplets are  $PE_{reusable} = \{ (1,1,1), (1,4,4), (2,2,2), (5, 2,1), (5,2,2), (5,2,3), (5,3,1) \}$ .

The remaining shaded entries in Table 4.26 are selected to meet coverage criteria. Hence  $PE' = \{ (1,4,3), (3,4,2), (3,4,3), (3,5,2), (3,7,1), (3,7,3), (4,5,3), (4,5,4) \}$ . This is only 53% of the failure mitigation testing requirements in [48]. Next, we generate new failure mitigation tests  $FMT'$  using the existing weaving rules that are defined in Table 3.7.

Table 4.27 shows the selected  $(i, p, e)$  triplets and resulting  $FMT'$  for the modified model  $BM'$ . The first column in Table 4.27 numbers each failure mitigation test ( $f_{mt_1} - f_{mt_9}$ ). The second column lists each  $(i, p, e)$  pair in  $PE'$ . The third column refers to the failure type whose mitigation is tested. The fourth column states the node at position  $p$ . The fifth column identifies the behavioral test used in constructing  $f_{mt'_i}$  ( $i = 1, \dots, 10$ ). The sixth column lists which  $mt_{ij}$  is used as described in Table 3.7. The seventh column shows the failure mitigation tests. The last column shows their length.



Table 4.27:  $FMT''$  for Modified Model  $BM'$ .

#	triplets $PE'$	Failure	Node	BT'' used	$mt_{ij}$ used	$FMT''$	Length
1	(1,4,3)	$f_3$	$n_2$	$bt'_1$	$mt_{31}$	$n_1, n_3, n_4, n_2, n_2$	5
2	(3,4,2)	$f_2$	$n_6$	$bt'_3$	$mt_{21}$	$n_1, n_3, n_5, n_6, n_1$	5
5	(3,4,3)	$f_3$	$n_6$	$bt'_3$	$mt_{31}$	$n_1, n_3, n_5, n_6, n_6, n_8, n_7, n_4, n_2$	9
6	(3,5,2)	$f_2$	$n_8$	$bt'_3$	$mt_{21}$	$n_1, n_3, n_5, n_6, n_8, n_1$	6
6	(3,7,1)	$f_1$	$n_4$	$bt'_3$	$mt_{11}$	$n_1, n_3, n_5, n_6, n_8, n_7, n_4, s_g$	8
6	(3,7,3)	$f_3$	$n_4$	$bt'_3$	$mt_{31}$	$n_1, n_3, n_5, n_6, n_8, n_7, n_4, n_4, n_2$	9
7	(4,5,3)	$f_3$	$n_7$	$bt'_4$	$mt_{31}$	$n_1, n_5, n_6, n_8, n_7, n_7, n_2$	7
8	(4,5,4)	$f_4$	$n_7$	$bt'_4$	$mt_{41}$	$n_1, n_5, n_6, n_8, n_7, s_1, s_2, n_2$	8
9	(4,5,4)	$f_4$	$n_7$	$bt'_4$	$mt_{42}$	$n_1, n_5, n_6, n_8, n_7, s_1, s_3, n_2$	8

#### 4.2.2.2 Changes to State-Event Matrix (SE)

Two types of changes can occur in  $SE'$ : (1) some failures become applicable in some states (changes from 0 to 1) or (2) not applicable (changes from 1 to 0). In many cases, the system requirements are changed for some states that impact some failure types to be feasible or infeasible for those states. For example, when the typed input (which can be incorrect) is replaced with button selection, an incorrectly typed input no longer occurs. Similarly, if power backup is provided, loss of power no longer is applicable. On the other hand, when new requirements are added that require mitigations where none were required before, this changes entries in the  $SE$  matrix from 0 to 1.

- Case A: Feasible to infeasible (changes from 1 to 0)

Let  $F_{inf}$  be the failure types that have become infeasible. Then  $E_{inf} = \{e | f_e \in F_{inf}\}$ . Any failure mitigation tests that were derived from a triplet  $(i, p, e)$  where the node in position  $p$  is a node for which the failure  $e$  is no longer applicable is now obsolete. These obsolete tests are given by  $FMT_o$ . Note that the failure mitigation test suite  $FMT \setminus FMT_o$  is reusable, not retestable, since we have run these tests already and they are not affected by the change. Back to our example in Subsection 3.1, assume failure type  $f_3$  is no longer

applicable in state  $n_6$ . From Table 3.6 and Table 3.9, we have three tests associated with  $f_3$  but only one test,  $f_{mt_{11}}$ , that uses  $n_6$ . Hence, only the test scenario (2,3,3) is obsolete. It was used to create  $f_{mt_{11}}$  which is now obsolete. Thus,  $FMT_o = \{f_{mt_{11}}\}$ .

Note that we do not have to rerun the remaining tests in Table 3.9, since they have already been executed.

- Case B: Infeasible to feasible (changes from 0 to 1)

This requires building a new potential failure scenario for failures that have become applicable. Let  $F_{se}$  be the subset of failure types that become now feasible such that  $F_{se} \subset F$ . Let  $S_{se}$  be the subset of states that become applicable for any failure  $f_j$  where  $f_j \in F_{se}$  and  $S_{se} \subset S$ .

Using our example, let failure type  $f_1$  become applicable in state  $n_6$  and  $f_4$  become applicable in state  $n_5$ . Thus,  $F_{se} = \{f_1, f_4\}$ , and  $S_{se} = \{n_5, n_6\}$ . The new  $SE'$  is defined in Table 4.28. It includes both making  $f_3$  inapplicable in state  $n_6$  as well as making failures  $f_1$  and  $f_4$  applicable in states  $n_5$  and  $n_6$  respectively.

Table 4.28: New State-Event Matrix SE'

Behavioral States/ Failure Type ( $f$ )	$n_1$	$n_2$	$n_3$	$n_4$	$n_5$	$n_6$	$n_7$
1	1	0	0	1	1	1	1
2	1	0	1	0	1	1	0
3	0	1	0	1	1	0	1
4	0	1	0	0	1	0	1

The new potential failure scenarios  $SP'$  and new selected triplets are based on the following:

- Remove tests from  $BT$  that do not visit states in  $S_{se}$  (states for which certain failure types have become applicable). The remaining behavioral

tests  $BT_{se}$  will be:  $BT_{se} = \{bt_i | \exists s \in S_{se} : bt_i \text{ visits } s\}$ .

Next, we build the potential failure scenario matrix based on  $BT_{se}$  and  $F_{se}$ . In the example, the affected node  $n_5$  exists only in  $bt_1$  and  $bt_2$ , and  $n_6$  exists only in  $bt_2$ . Thus,  $BT_{se} = \{bt_1, bt_2\}$ .

- Remove all failures that are not in  $F_{se}$ . In our example, we exclude  $f_2$  and  $f_3$ .
- Rebuild the new potential failure scenarios as  $SP'$ . Table 4.29 shows the new potential failure scenarios  $SP'$  marking each feasible entry as a "1". Since we are not concerned with nodes other than  $n_5$  and  $n_6$ , entries for other nodes are marked with 'x'.

Table 4.29: The New Potential Failure Scenarios  $SP'$

i	1				2				
	$bt_1$				$bt_2$				
$BT_{se}$	1	2	3	4	1	2	3	4	5
F/N	$n_1$	$n_5$	$n_7$	$n_2$	$n_1$	$n_5$	$n_6$	$n_7$	$n_2$
$f_1$	x	x	x	x	x	x	1	x	x
$f_4$	x	1	x	x	x	1	x	x	x

- Select a new set of triplets  $PE'$ . Table 4.29 shows these as shaded entries.  $PE' = \{(1, 2, 4), (2, 3, 1)\}$ .

There are no reusable triplets. Note, that the definition of  $PE'$  requires to select as test requirements all occurrences of nodes  $s \in S_{se}$ . If such nodes occur many times, it may be useful to restrict the number of times that position  $p$  is selected where  $node(p) \in S_{se}$ . We can use coverage criteria.

- Generate new failure mitigation tests  $FMT''$  as in Subsection 3.7. Table 4.30 shows the selected triplets and new failure mitigation regression

tests.

Subsection 4.2.2.4 describes the case when there are multiple changes to the artifacts  $(BM, SE, F)$ .

Table 4.30: Selected Triplets and Constructing  $FMT''$  for  $F_{se}$ .

#	Selected triplets ( $PE'$ )	Failure	Node	BT used	$mt_{ij}$ used	$FMT''$	Length
1	(1,2,4)	$f_4$	$n_5$	$bt_1$	$mt_{41}$	$n_1, n_5, s_1, s_2, n_5, n_7, n_2$	7
2	(1,2,4)	$f_4$	$n_5$	$bt_1$	$mt_{42}$	$n_1, n_5, s_1, s_3, n_5, n_7, n_2$	7
3	(2,3,1)	$f_1$	$n_6$	$bt_2$	$mt_{11}$	$n_1, n_5, n_6, s_g$	4

#### 4.2.2.3 Changes to Failure Types (F)

First, we assume that the changes to failure types are the only changes. Later, we will discuss the situation when changes to multiple artifacts occur. We describe the change in failure type as follows:

- Delete failure types  $F_d = \{f_{d_1}, \dots, f_{d_m}\}$  where  $m$  is the number of deleted failure types:  $F' = F \setminus F_d$

An example of deleting a failure type: a faulty network can no longer cause a network connection error by using a backup router to quickly swap out the faulty network. Suppose, we delete the failure types ( $f_2$  and  $f_3$ ) from our example. Thus,  $F_d = \{f_2, f_3\}$ .

Next, we remove the mitigation models  $\{MM_{d_1}, \dots, MM_{d_m}\}$  and weaving rules  $\{WR_{d_1}, \dots, WR_{d_m}\}$ . Any failure mitigation tests that test proper mitigation for a failure  $f \in F_d$  is removed as well. Let  $FMT_{F_d} \subseteq FMT$  such that each  $t \in FMT_{F_d}$  was constructed to test a failure type  $f \in F_d$ .  $FMT_{F_d} = \{fmt | fmt \in FMT \wedge fmt \text{ based on pair } (bt, p, e') \in PE \text{ where } f_{e'} \in F_d \text{ and } 1 \leq p \leq Length(btc)\}$ . Note that  $FMT \setminus FMT_{F_d}$  is reusable, not retestable, since we have executed these tests already.

Back to our example, we remove the mitigation models  $MM_{F_d} = \{MM_2, MM_3\}$  and weaving rules  $WR_{F_d} = \{WR_2, WR_3\}$ . From Table 3.9, the deleted failure mitigation tests are  $FMT_{F_d} = \{fmt_2, fmt_3, fmt_6, fmt_8, fmt_9\}$ . Table 4.31 shows the reusable tests after deleting failures  $f_2$  and  $f_3$ . We do not have to rerun these tests.

Table 4.31: Reusable Tests After Deleting Failures  $f_2$  and  $f_3$

#	Selected triplets (PE)	Failure	Node	BT used	$mt_{ij}$ used	FMT	Length
1	(1,2,1)	$f_1$	$n_5$	$bt_1$	$mt_{11}$	$n_1, n_5, s_g$	3
2	(2,8,4)	$f_4$	$n_7$	$bt_2$	$mt_{41}$	$n_1, n_5, n_6, n_7, s_1, s_2, n_7, n_2$	8
3	(2,8,4)	$f_4$	$n_7$	$bt_2$	$mt_{42}$	$n_1, n_5, n_6, n_7, s_1, s_3, n_2$	7
4	(2,3,1)	$f_1$	$n_7$	$bt_2$	$mt_{11}$	$n_1, n_5, n_6, n_7, s_g$	5

- Add new failure type  $F_a = \{f_{a_1}, \dots, f_{a_n}\}$  where  $n$  is the number of new failure types:

In this case, we need to build a state-event matrix for the new failure, construct a potential failure scenario matrix for the test suite and the new failures, and generate (i,p,e) triplets. We also must define mitigation models and weaving rules for the new failures and create failure mitigation tests for the failure scenarios. Note that we do not need to rerun  $FMT$ , nor include existing failure types in the construction of the potential failure scenarios. Thus, a new state-event matrix  $SE_a$  is an  $n \times |S|$  matrix where

$$SE_a(i, j) = \begin{cases} 1 & \text{if failure type } j \text{ applies in node } n_i \in S \\ 0 & \text{otherwise} \end{cases}$$

Note that the number of states  $i = 1, \dots, |S|$  and number of failures types  $j = 1, \dots, n$ .

Using our example, we add two new failure types ( $f_5$  and  $f_6$ ), so  $F_a = \{f_5, f_6\}$ .

Next, we have to build a new state-event matrix ( $SE_a$ ) that includes the new failure types  $F_a$  as shown in Table 4.32.

Table 4.32: New State-Event Matrix  $SE_a$

Behavioral States/ Failure Type ( $f$ )	$n_1$	$n_2$	$n_3$	$n_4$	$n_5$	$n_6$	$n_7$
5	0	0	0	0	0	1	0
6	0	1	0	0	0	0	1

Next, we create new mitigation models  $MM_a = \{MM_{a1}, \dots, MM_{an}\}$  for each failure type in  $F_a$ . Let  $WR_a = \{WR_{a1} \dots WR_{an}\}$  be the weaving rules for the new mitigation models. Next, we generate the new mitigation test suites  $MT_a = \{MT_{a1}, \dots, MT_{an}\}$ . In the example, we add new mitigation models  $MM_a = \{MM_5, MM_6\}$  and new weaving rules ( $WR_5$  and  $WR_6$ ) for the new failure types. They are shown in Table 4.33. The first mitigation model returns to state  $n_6$  after a database error, the second returns to the start node  $n_1$  and ends the test.

Next, we create new mitigation test paths  $MTs_a$  using the new mitigation models. In our case, there are no mitigation test paths. Applying the weaving rule is the only task required (c.f Table 4.33).

Then, we build the new potential failure scenario matrix. The new potential failure scenario matrix is defined as follows:  $SP_a = \{(i, p, e) | 1 \leq i \leq l, 1 \leq p \leq len(bt_i), 1 \leq e \leq n, SE_a(node(p), e) = 1\}$ . We select a new set of triplets  $(i, p, e) \in PE_a$  based on coverage criteria. How we select the  $(i, p, e)$  triplets depends on the size of the new potential failure scenario.

In the example, the new potential failure scenario is shown in Table 4.34. Using our coverage criteria results in selecting three triplets as shown in Table 4.35.

Table 4.33: New Mitigation Requirements

MM	Explanation	Model	WR#
MM5	Retry: database server error	$MT_5 = \phi$ where $node(p)^r$ is the state in which we are trying to save data $node(p) = n_6$ and $r = 1$ (retry once)	2
MM6	End Activity: misunderstood behaviour such as try to access node without having specific user role	$MT_6 = \phi$ where $s_f = n_1$ , and $s_f$ is the start node and stop	4

Table 4.34: New Potential Failure Scenario  $SP_a$  Due to the Added Failures  $F_a$

i	1				2					3				
BT	$bt_1$				$bt_2$					$bt_3$				
	1	2	3	4	1	2	3	4	5	1	2	3	4	5
	$n_1$	$n_5$	$n_7$	$n_2$	$n_1$	$n_5$	$n_6$	$n_7$	$n_2$	$n_1$	$n_3$	$n_4$	$n_3$	$n_2$
$f_5$	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$f_6$	0	0	1	1	0	0	0	1	1	0	0	0	0	1

Finally, new failure mitigation tests  $FMT_{F_a}$  are generated from triplets in  $PE_a$ . Hence,  $FMT'' = FMT_{F_a}$ .

In the example, new failure mitigation tests  $FMT_{F_a}$  are generated based on new weaving rules  $WR_5$  and  $WR_6$  for the new failure types in  $F_a = \{f_5, f_6\}$ . Table 4.35 shows the selected triplets and  $FMT''$ .

Table 4.35: Selected Triplets and Resulting  $FMT_{F_a}$

#	Selected triplets (PE)	Failure	Node	BT used	$mt_{ij}$ used	$FMT_{F_a} = FMT''$	Length
1	(1,3,6)	$f_6$	$n_7$	$bt_1$	$mt_{61}$	$n_1, n_5, n_7, n_1$	4
2	(2,3,5)	$f_5$	$n_6$	$bt_2$	$mt_{51}$	$n_1, n_5, n_6, n_6, n_7, n_2$	6
3	(3,5,6)	$f_6$	$n_2$	$bt_3$	$mt_{61}$	$n_1, n_3, n_4, n_3, n_2$	5

So far, we assumed that the addition of new failures does not impact existing failure mitigations. Otherwise, we would need to include mitigation tests for failures that have been affected by mitigation of new failures. This can happen when the mitigations for two different failure types share some of the mitigation code. Let impacted failures types  $F_c = \{f_1, \dots, f_q\}$  be the failures impacted by new mitigation requirements, where  $q$  is the number of affected failures. Let the impacted failure mitigation tests  $FMT_c = \{fmt | fmt_j \text{ built from pair } (i, p, e) \in PE \text{ where } f_e \in F_c\}$ . The new failure mitigation tests derived from  $F_c$  will be as follows:  $FMT'' = FMT_{F_a} \cup FMT_c$ .

Note:  $FMT$  does not have to be rerun and  $FMT_c$  is a selection not a creation of new tests.

In our example, suppose that mitigation of new failures impacts existing failure mitigation. Let failure type  $f_1$  be impacted by new mitigation requirements. As a result,  $F_c = \{f_1\}$ , and from Table 4.31 (after deleting failure mitigation tests based on deleted failures  $FMT_{F_d}$ ), the impacted failure mitigation tests are  $FMT_c = \{fmt_1, fmt_4\}$ . The new failure mitigation tests  $FMT''$  consist of those in Table



4.31 plus  $FMT_c$ .

#### 4.2.2.4 Changes to Behavioral Model (BM), Failure Types (F) and State Event Matrix (SE)

When  $BM$ ,  $F$  and  $SE$  have been changed at the same time, we need to perform the following steps:

- Classify  $BT$  into  $BT_r$ ,  $BT_u$ , and  $BT_o$  as in section 4.2.2.1.1.
- Generate new behavioral test cases  $BT'$  due to the changes to  $BM$ .
- Create new mitigation models  $MM_{aj}$ ; add the weaving rules for the new failures  $F_a$ , and generate new mitigation tests  $MT_{aj}$  for new failure types  $F_a$ .
- Delete the mitigation models and weaving rules, and failure mitigation tests  $FMT_{F_d}$  for all deleted failures in  $F_d$ .
- Remove any obsolete mitigation tests  $FMT_o$  due to the changes to  $SE$  for any inapplicable failure types.
- Define the subset of failure types  $F_{se}$  and subset of states  $S_{se}$  that are affected and become now feasible due to the changes to  $SE$ .
- Build a new state-event matrix  $SE'$  that includes any new states due to the changes to  $BM$  and the added failure types due to the changes to  $F$ . Also, because of the changes to  $SE$ , the new state-event matrix  $SE'$  includes any failure types in  $F_{se}$ .
- Construct new potential failure scenarios  $SP'$  as follows :
  - Use  $BT'' = BT' \cup BT_r$  as in 4.2.2.1.3.

- Use the current failure types  $F' = (F \setminus F_d) \cup F_a$ . That is we remove deleted failures and add new failures for the rows in SP' (Section 4.2.2.3).
- Construct the current SE' matrix by applying any changes to applicability for existing nodes/failures, adding new rows for new failures and new columns for new nodes (Section 4.2.2.2, 4.2.2.3) and removing columns for deleted nodes.
- SP' is then constructed based on F' and SE'.

Because of the changes to  $SE$ ,  $f_1$  becomes applicable in state  $n_6$  and  $f_4$  becomes applicable in state  $n_5$ . Table 4.36 shows these changes as shaded '1'. It also has 2 new columns for newly added node  $n_8$ . Thus,  $F_{se} = \{f_1, f_4\}$ , and  $S_{se} = \{n_5, n_6\}$ . In our example, BT'' is as before. It's concatenation is shown in row 3 of Table 4.37. In the example, we delete the failure types ( $f_2$  and  $f_3$ ), and add new failure types ( $f_5$  and  $f_6$ ). Thus,  $F_d = \{f_2, f_3\}$ , and  $F_a = \{f_5, f_6\}$ . Hence  $F = \{f_1, f_4, f_5, f_6\}$ . These are shown in column 1 of Table 4.37.

Table 4.36: New State-Event Matrix  $SE'$

Behavioral States/ Failure Type ( $f$ )	$n_1$	$n_2$	$n_3$	$n_4$	$n_5$	$n_6$	$n_7$	$n_8$
1	1	0	0	1	1	1	1	0
4	0	1	0	0	1	0	1	0
5	0	0	0	0	0	1	0	0
6	0	1	0	0	0	0	1	1

Table 4.37: New Potential Failure Scenarios  $SP'_1$  ( $BM$  Change and  $f_2, f_3$  Deleted)

	1				2						3								4						5			
	$bt'_1$				$bt'_2$						$bt'_3$								$bt'_4$						$bt_1$			
Position (P)	1	2	3	4	1	2	3	4	5	6	1	2	3	4	5	6	7	8	1	2	3	4	5	6	1	2	3	4
$I'/F$	$n_1$	$n_3$	$n_4$	$n_2$	$n_1$	$n_3$	$n_5$	$n_7$	$n_4$	$n_2$	$n_1$	$n_3$	$n_5$	$n_6$	$n_8$	$n_7$	$n_4$	$n_2$	$n_1$	$n_5$	$n_6$	$n_8$	$n_7$	$n_2$	$n_1$	$n_5$	$n_7$	$n_2$
$f_1$	1	0	1	0	1	0	1	1	1	0	1	0	1	1	0	1	1	0	1	1	1	0	1	0	1	1	1	0
$f_4$	0	0	0	1	0	0	1	1	0	1	0	0	1	0	0	1	0	1	0	1	0	0	1	1	0	1	1	1
$f_5$	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$f_6$	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0	1	1	1	0	0	1	1

The next step is to select a new set of triplets from the potential failure scenarios. Tables 4.37 marks triplets that meet the coverage criteria according to [48]. There are 12 triplets. However, three of the triplets are reusable and can be removed: (1,1,1), (1,4,4), (5,3,1), resulting in a 25% saving over [48]. (The corresponding 12 FMT' are shown in Table 4.38).

Table 4.38: Constructing  $FMT'$  with  $PE'$ .

#	triplets $PE'$	Failure	Node	BT' used	$mt_{ij}$ used	$FMT'_1$	Length
1	(2,5,1)	$f_1$	$n_4$	$bt'_2$	$mt_{11}$	$n_1, n_3, n_5, n_7, n_4, s_g$	6
2	(3,3,1)	$f_1$	$n_5$	$bt'_3$	$mt_{11}$	$n_1, n_3, n_5, s_g$	4
3	(3,3,4)	$f_4$	$n_5$	$bt'_3$	$mt_{41}$	$n_1, n_3, n_5, s_1, s_2, n_5, n_6, n_8, n_7, n_4, n_2$	11
4	(3,3,4)	$f_4$	$n_5$	$bt'_3$	$mt_{42}$	$n_1, n_3, n_5, s_1, s_3, n_5, n_6, n_8, n_7, n_4, n_2$	11
5	(3,4,1)	$f_1$	$n_6$	$bt'_3$	$mt_{31}$	$n_1, n_3, n_5, n_6, s_g$	5
6	(3,4,5)	$f_5$	$n_6$	$bt'_3$	$mt_{51}$	$n_1, n_3, n_5, n_6, n_6, n_8, n_7, n_4, n_2$	9
7	(4,4,6)	$f_6$	$n_8$	$bt'_4$	$mt_{61}$	$n_1, n_5, n_6, n_8, n_1$	5
8	(4,5,4)	$f_4$	$n_7$	$bt'_4$	$mt_{41}$	$n_1, n_5, n_6, n_8, n_7, s_1, s_2, n_7, n_2$	9
9	(4,5,4)	$f_4$	$n_7$	$bt'_4$	$mt_{42}$	$n_1, n_5, n_6, n_8, n_7, s_1, s_3, n_7, n_2$	9
10	(5,3,6)	$f_6$	$n_7$	$bt_1$	$mt_{61}$	$n_1, n_5, n_7, s_g$	4
11	(5,4,4)	$f_4$	$n_2$	$bt_1$	$mt_{41}$	$n_1, n_5, n_7, n_2, s_1, s_2, n_2$	7
12	(5,4,4)	$f_4$	$n_2$	$bt_1$	$mt_{42}$	$n_1, n_5, n_7, n_2, s_1, s_3, n_2$	7

#### 4.2.2.5 Changes to Mitigation Models (MM)

We assume that  $BT', F', SE'$ , and  $PE'$  have been constructed based on changes to corresponding artifacts and algorithms in Subsections 4.2.2.1, 4.2.2.2, 4.2.2.3, and 4.2.2.4. We also assume that  $MT'_j$  have been constructed for any changed model:  $j \in mod_{MM} = \{j | 1 \leq j \leq |F| \wedge MM'_j \neq MM_j\}$ . When changes to mitigation models occur, we need to do the following:

- determine mitigation test paths for the changed mitigation model(s).
- determine retestable  $FMT'$  and triplets upon which they are based that use mitigation tests from the changed model(s).

Note that we do not have to consider  $PE'_2$  or  $PE'_3$  since they are based on new mitigation models, not changed ones, or failure types that used to be inapplicable,

hence no retestable failure mitigation tests exist. Note that  $SE'_3$  only specifies feasible triplets that did not exist in the prior version. Note also that  $PE'_1$  describes new test requirements from which new failure mitigation tests are created (regardless of changes to the mitigation models).

Hence we only need to consider two cases:

- failure mitigation tests that are based on retestable mitigation tests. These need to be rerun.
- failure mitigation tests that were built based on failure types for the modified mitigation model. These need to be rebuilt with the new mitigation test.

We remove any obsolete failure mitigation tests  $FMT_o$ .

#### 4.2.2.5.1 Determine Mitigation Test Paths for Changed Mitigation Models

Since the mitigation model is similar to the behavioral model (FSMWeb), the same concept is applied to the mitigation test paths using Andrews et al. [49]. The same classification will be used in terms of obsolete, reusable, or retestable test paths. We classify mitigation tests ( $MT_j$ ) of failure type  $f_j$  as obsolete ( $MT_{j_o}$ ), retestable ( $MT_{j_r}$ ), and reusable ( $MT_{j_u}$ ). We determine new mitigation test cases ( $MT'_j$ ) for uncovered edges in the mitigation model of failure type  $f_j$ .

Back to our example, the mitigation model  $MM_4$  for failure type  $f_4$  is modified as shown in Figure 4.7. We modify the model from export to Excel to be exported to Word format. We delete edges:  $(s_1, s_2), (s_2, s_f)$ . The deleted edges make mitigation test  $mt_{41}$  obsolete. Thus,  $MT_{4o} = \{mt_{41}\}$ . We also add node  $(s_4)$  and edges:  $(s_1, s_4), (s_4, s_f)$ . As a result, a new mitigation test path is needed:  $mt'_{41} = \{s_i, s_1, s_4, s_f\}$ . Thus,  $MT'_4 = \{mt'_{41}\}$ . Since mitigation test  $mt_{42} = \{s_i, s_1, s_3, s_f\}$  visits  $s_i$ , this makes  $mt_{42}$  retestable. Consequently,  $MT_{4r} = \{mt_{42}\}$ . Since only one

mitigation model has been changed,  $mod_{MM} = \{4\}$ . The new mitigation test for  $MM'_4$  is  $MT''_4 = MT_{4r} \cup MT'_4 = \{mt_{42}, mt'_{41}\}$ .

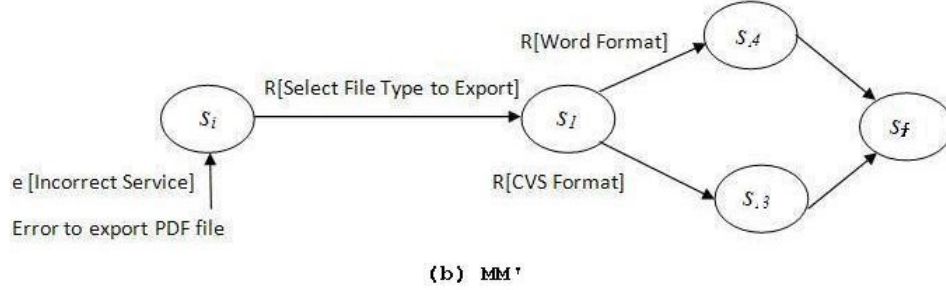


Figure 4.7: Modified Mitigation Model  $MM4$  or  $MM6$ .

#### 4.2.2.5.2 Failure Mitigation Tests Based on Retestable Mitigation Tests

As stated before these tests need to be rerun. They constitute the retestable failure mitigation tests  $FMT_r$ . They are defined as follows:  $FMT_r = \{fmt | fmt \in FMT \text{ based on: } (i, p, j) \in PE, j \in mod_{MM}, mt_j \in MT_{jr}\}$ . We need to rerun  $FMT_r$ . From Table 3.9, the only failure mitigation tests used for  $f_4$  are  $fmt_{13}$ ,  $fmt_{14}$ ,  $fmt_{15}$  and  $fmt_{16}$ . However,  $fmt_{13}$  and  $fmt_{15}$  are obsolete because of using  $mt_{41}$ , and  $fmt_{14}$  and  $fmt_{16}$  become retestable because of using  $mt_{42}$ . Hence  $FMT_r = \{fmt_{14}, fmt_{16}\}$ .

#### 4.2.2.5.3 Build New Failure Mitigation Tests

Here we need to make sure that all  $mt_j \in MT'_j$  ( $j \in mod_{MM}$ ) are used to build new failure mitigation tests. This requires identifying all triplets:  $PE_{MM'} = \{(i, p, j) | j \in mod_{MM}\}$  and then using  $mt_j \in MT'_j$  to build the new failure mitigation tests  $FMT'_{MM}$ .  $FMT'_{MM} = \{fmt' | \text{based on } (i, p, e) \in PE_{MM'} \text{ using}$

$mt_j \in MT'_j, j \in mod_{MM}\}$ . These new tests need to be run.

In the example and from Table 3.9,  $PE_{MM'} = \{(1, 3, 4), (1, 4, 4)\}$  and  $MT'_4 = \{mt'_{41}\}$ . We generate new failure mitigation tests using  $mt'_{41}$ . Thus,  $fmt'_{13} = \{n_1, n_5, n_7, s_1, s_4, n_2\}$ , and  $fmt'_{15} = \{n_1, n_5, n_7, n_2, s_1, s_4, n_2\}$ .

#### 4.2.2.5.4 Potential Impact on Other Failure Mitigations

Changes to failure mitigations can impact mitigations of other failures whose models have not changed. This can happen when they share portions of the mitigation code, for example. In such a case, the failure mitigation tests for these failures need to be rerun. Let  $E_I$  be the failure types impacted. Then all failure mitigation tests based on  $PE_I = \{(i, p, e) | e \in E_I\}$  need to be rerun. The failure mitigation tests  $FMT_I = \{fmt | fmt \in FMT, \text{ based on } (i, p, e) \in PE_I\}$  are the impacted set of tests.

From the example, we assume the changes to  $MM'_4$  have affected the failure mitigation associated with failure type  $f_1$ . Using Table 3.9,  $E_I = \{1\}$ ,  $PE_I = \{(1, 3, 4), (1, 4, 4)\}$  and  $FMT_I = \{fmt_1, fmt_7\}$ . These need to be rerun.

In summary, the regression test suite  $FMT''$  to execute consists of retestable tests  $FMT_{rt}$ , new failure mitigation tests  $FMT'_{MM}$  and tests for failures that are impacted by mitigation model changes in other failures  $FMT_I$ . Hence  $FMT'' = FMT_r \cup FMT'_{MM} \cup FMT_I = \{fmt_{14}, fmt_{16}\} \cup \{fmt'_{13}, fmt'_{15}\} \cup \{fmt_1, fmt_7\}$

#### 4.2.2.6 Changes to Weaving Rules (WR)

When a weaving rule is modified, we need to identify which  $(i, p, e)$  triplets are affected and we need to regenerate tests that were created using the old weaving rule, as these failure mitigation tests are now obsolete. Let  $mod_{WR} = \{\text{failure types for which weaving rule changed}\}$ . Let  $PE_{WR} = \{(i, p, e) | (i, p, e) \in PE \wedge e \in$

$mod_{WR}\}$ . We generate new failure mitigation tests for these  $(i, p, e)$  triplets. Then  $FMT' = \{fmt' | (i, p, e) \in PE_{WR} \text{ used to construct } fmt'\}$ . We simply regenerate failure mitigation tests using appropriate coverage criteria and rerun these tests.

In the example, we update the weaving rule "End All" for failure  $f_2$  to be "Fix and proceed". Hence  $PE_{WR} = \{(1, 2, 2), (2, 3, 2), (3, 2, 2)\}$  (cf. Table 3.9). Tests  $fmt_2$ ,  $fmt_6$  and  $fmt_7$  are obsolete and need to be regenerated. Table 4.39 shows the result. Using Table 3.9, we use the new weaving rule  $wr'_{f_2}$  and generate new failure mitigation tests  $FMT''$  for them using the new weaving rule for triplets in  $PE_{WR}$ . The new failure mitigation tests are shown in Table 4.39.

In case of multiple changes to artifacts, we do not have to consider  $PE'_1$  or  $PE'_3$ , since they are based on new mitigation models (and weaving rules), not changed ones or failures that used to be inapplicable, hence no retestable failure mitigation tests exist. Since  $PE'_1$  describes new test requirements, we would have already constructed failure mitigation tests with the new weaving rules as described in Subsection 4.2.2.4.  $FMT'$  refers to reconstruction of existing tests.

Table 4.39:  $PE_{WR}$  and Resulting  $FMT$ .

#	triplets (PE)	Failure	Node	BT used	$mt_{ij}$ used	FMT''	Length
1	(1,2,2)	$f_2$	$n_5$	$bt_1$	$mt_{21}$	$n_1, n_5, n_5, n_7, n_2$	5
2	(2,3,2)	$f_2$	$n_6$	$bt_2$	$mt_{21}$	$n_1, n_5, n_6, n_6, n_7, n_2$	6
3	(3,2,2)	$f_2$	$n_3$	$bt_3$	$mt_{21}$	$n_1, n_3, n_3, n_4, n_3, n_2$	6

### 4.2.3 Discussion

We built the framework for selective regression testing of fail-safe behavior. We provide a systematic method by showing the formalization steps for each type of change to the various models (BM, SE, F, MM, WR) and build a regression test suite based on each type of change, allowing for multiple changes to artifacts. We use

coverage criteria [48] to construct the new test requirements. We improve efficiency by removing reusable test requirements. In case changes to the behavioral model are the only change and the changes did not add many new states/edges or delete many states/edges, there will be few new mitigation tests. When failure types no longer apply or when some failure types become not applicable in some nodes (entries in  $SE$  matrix change from 1 to 0), the number of failure mitigation tests becomes smaller. However, when new failure types are required to be mitigated, or some failure types become applicable in states where they were not applicable before, the number of failure mitigation tests increases. Therefore, the changes to behavioral models  $BM$ , the changes to  $SE$ , or adding new failure types have higher impact as more work is involved such as building  $SE'$ , potential failure scenarios  $SP$ , and generating more failure mitigation tests. On the other hand, changes to mitigation models or weaving rules have less impact because they are local changes and occur late in the failure mitigation test generation process. Multiple changes to the artifacts can become expensive and may require to create a full new test suite.

Table 4.40 summarizes and compares the effects of the various changes and combinations of changes to our example. The top half of the table reports changes to  $BM$ ,  $SE$ , and  $F$  while the bottom will discuss multiple changes to all three artifacts. For phase 1, we list the number of retestable behavioral tests  $|BT_r|$ , the number of retestable failure mitigation tests  $|FMT_r|$ , and the number of new behavioral tests  $|BT'|$ , for phase 2, the length of  $BT''$ , as well as the number of failure scenario triplets and the number of reusable ones. For Phase 3, we list the number of failure mitigation tests in the new test suite  $|FMT''|$  and their length. The top half of columns 2 and 3 compares selective regression testing (SR) against a full new test path suite (FR) for changes to  $BM$  only. The top half of columns 4 and 5 compares selective regression testing (SR) and a full retest (FR) for changes to  $SE$ . The re-



maining pairs of columns compare SR and FR with respect to changes to F, and changes to failures that may impact other failures (F with impact). Row 10 and 20 of Table 4.40 show the proportion of the length of the test suite for selective regression testing compared to a full retest. The second half of Table 4.40 reports results based on multiple changes to BM, SE and F as well as changes to mitigation models (with and without impact) and weaving rules. Results are reported for selective regression testing (SR) and a full retest (FR), so a comparison can be made.

Table 4.40: Effect of Changes Using Selective Regression Testing (SR) and Full Retest (FR)

	Phase #	BM		SE		F		F with impact	
		SR	FR	SR	FR	SR	FR	SR	FR
1	$ BT_r $	1	0	2	0	3	0	2	0
	$ FMT_r $	0	0	0	0	0	0	4	0
	$ BT' $	4	0	0	0	0	0	0	0
2	Length (BT <sup>m</sup> )	28	27	9	14	14	14	14	14
	PE Triplets	8	15	2	16	3	9	3	9
	Reusable Triplets	7	0	0	0	0	0	0	0
3	$ FMT'' $	10	17	3	19	3	11	7	11
	Length	65	99	18	96	15	56	28	56
4	SR/FR (%)	66		19		27		50	
	Phase #	BM, SE, F		MM		MM with impact		WR	
		SR	FR	SR	FR	SR	FR	SR	FR
1	$ BT_r $	1	0	1	0	2	0	3	0
	$ FMT_r $	0	0	0	0	4	0	0	0
	$ BT' $	4	0	0	0	0	0	0	0
2	Length (BT <sup>m</sup> )	28	27	4	14	4	14	14	14
	PE Triplets	9	12	2	14	2	14	3	14
	Reusable Triplets	3	0	0	0	0	0	0	0
3	$ FMT'' $	12	15	4	16	8	16	3	16
	Length	87	90	26	82	39	82	17	86
4	SR/FR (%)	97		32		48		20	

Table 4.40 shows that changing the behavioral model alone or in conjunction with other changes requires the most regression testing, but is still no more than 97% of a full retest. When changes are made to the behavioral model alone (columns 2/3 in the upper half of Table 4.40) selective regression test is shorter (65 vs. 99). The length of the regression test suite is the second highest. This is because we added two nodes and deleted one edge, a relatively large change for a small model.

This increased the number of possible failure scenarios. Changing applicability, failure types, mitigation models or weaving rules favors selective regression testing in all cases. Changing SE affects failure  $f_4$  and because of multiple mitigation paths results in two new failure mitigation tests for each failure scenario. Yet we still only have 19% of the length of a full retest. Changes to F deleted two failures and added two new ones. Depending on whether failures impact others, selective regression testing needs between 27% and 50% of a full retest. The smallest proportion of tests occurs for changes in weaving rules (20%). This makes sense as weaving is the last step in the test path generation for failure mitigation tests. Overall, a full retest is always a lot more expensive in this example. Selective regression testing can be as small as 19% of a full retest. Note that changes to the behavioral model can and do result in reusable triplets. Removing them results in 47%, and 25% reduction in test requirements respectively, of the mitigation testing requirements Andrews et al. [48] would have used. For larger models, this can result in significant savings due to fewer failure mitigation tests that need to be generated and executed. We will explore this further in the case study of Section 4.3.

#### 4.2.4 Comparison to Earlier Work

Table 4.41 summarizes contributions of earlier works. Boukhris *et al.* [70] propose an approach that leverages a test suite derived from an FSMWeb model of a web application [51] and transforms it into a series of mitigation tests for various failures. A similar approach was defined for generating safety-mitigation tests for safety-critical systems [48]. The two approaches differ in the models used and the rules to generate mitigation test requirements. They describe an approach that

- uses a primary behavioral model and its associated behavioral test suite,

- considers a set of defined failures and how they need to be mitigated,
- generates a set of mitigation tests for each failure type,
- uses failure mitigation coverage criteria [48] or a genetic algorithm [70] to determine where in the behavioral test suite a failure should occur (fail-safe test requirement).
- creates fail-safe behavior tests based on these failure scenarios.

Table 4.41: Comparison of Approaches

Reference	External Failure Testing	Selective Regression Testing	Coverage Criteria	Model	Eliminate Reusable Failure Scenarios	Domain
A case study of black box fail-safe testing in web applications [70]	YES	NO	NO	FSMWeb	NO	Web Application
Testing Web applications by modeling with FSMs [51]	NO	NO	YES	FSMWeb	NO	Web Application
Fail safe test generation in safety critical systems [48]	YES	NO	YES	CEFSM	NO	Safety-Critical
Selective regression testing of safety-critical systems: a black box approach [47]	YES	YES	YES	EFSM	NO	Safety-Critical

Neither of these discusses selective regression testing. Boukhris *et al.* [70] do not address regression testing. However, they treat primary behavior, failures, and mitigation distinctly in the stages described above. Thus the approach is naturally suited for partial regeneration of tests after changes in behavior, failures, or failure mitigation requirements. Subsections 4.2.1, 4.2.2, and 4.2.3 provide this extension. Depending on where in the process changes occur, only a subset of safety mitigation test generation stages will have to be repeated. We also define rules on how to do this. The approach presented here also extends Boukhris *et al.* [70] by providing an alternate way (coverage criteria instead of a genetic algorithm) to generate mitigation test requirements. This is important because the simulation study in Boukhris *et al.* [69] found that the genetic algorithm is inferior to coverage criteria when the search space (i.e. the potential failure scenarios) is small. This happens when changes are limited.

We also extend work in Andrews *et al.* [47] as follows:

- First, we use a different behavioral model (FSMWeb, instead of EFSM) and (regression) test generation technique for it. Note also, that Andrews *et al.* [47] does not use clusters in the model and generates behavioral test differently due to the reachability issues inherent in (C)EFSMs.
- Second, we use a different domain (web applications rather than safety-critical systems).
- Third, we use external failures and mitigation requirements common to web applications rather than safety-critical systems.
- Fourth, we formalize in detail the regression testing and partial test regeneration procedure.

- Fifth, we remove inefficiencies by identifying reusable failure scenarios and not requiring them to be tested again. As we saw in the example (rows 6/7 and 16/17) this led to a removal of 47% (of 15 triplets, 7 are reusable, 8 are required) of reusable test requirements for behavioral changes and 25% (of 12 triplets, 3 are reusable, 9 are required) in the case of multiple types of changes compared to using Andrews et al. [48] or [47]. As we will see in the case study in Subsection 4.3, Table 4.42, efficiency is improved by 65% and 45%, respectively.
- Sixth, we present a large case study of a mortgage system at one of our industrial partners, a major lending firm to show applicability to a large commercial web application.

Table 4.41 summarizes the differences.

## 4.3 Mortgage System Case Study: Regression Test

### 4.3.1 Case Study Objectives

Boukhris et al. [70] use a mortgage system as a case study for their approach to test fail-safe behavior. They explored applicability, scalability, effectiveness and efficiency. Here, we propose to investigate the efficiency of our proposed selective regression testing approach by defining changes to the system and comparing selective regression testing to a full retest of the system. Unlike Boukhris et al. [70], we do not use a genetic algorithm to generate failure scenarios. Instead we use the coverage criteria introduced earlier.

### 4.3.2 Case Study Research Questions

1. RQ1: What are the results when we apply criteria-based selective regression testing to a large web application? So far, coverage criteria together with a selective regression testing approach in Andrews et al. [47] have only been applied to a tiny example with 4 nodes and 6 edges (and a different behavioral model that does not use a hierarchical approach with clusters). What are the results for a large web application?
2. RQ2: Are there any efficiencies to be obtained compared to a full retest of the application? If so, what are they?
3. RQ3: Would it have been possible to apply a genetic algorithm instead of coverage criteria?
4. RQ4: How many tests can be omitted by eliminating reusable failure scenarios?

### 4.3.3 General Description

We focus on the Closing Documents subsystem (CD) as shown in the top of Figure 4.4 in Subsection 4.1.2. Figure 4.4 shows the behavioral model for CD before and after changes.

We now make the following changes:

1. Changes to the behavioral model: we add a new funding page and remove the web page that shows past closing instructions (See the bottom of Figure 4.4).

2. Changes to the applicability matrix: we assume that  $f_7$  becomes in-applicable on the web page that shows documents to close (DC) in the CD cluster and becomes applicable for the web page that shows advanced search (AS) results in the Search cluster.
3. Changes to external failure types: first, we delete  $f_1$  as a faulty network can no longer cause a network connection error by using a backup router to quickly swap out the faulty network, and  $f_2$  assuming session expiration failure is no longer valid. Then, we add a mitigation requirement for power outage (failure  $f_{11}$ ) (end activity).
4. Changes to mitigation models: first, we modify  $MM_4$  from export to Excel to be exported to Word format as shown in Figure 4.5. Then, we assume the change to  $MM_4$  has affected the failure mitigation associated with failure type  $f_1$ .
5. Changes to weaving rules: we update the weaving rule "End All" for failure  $f_2$  to be "Fix and proceed".

First, we apply each of the above changes separately, then we apply them together. Table 4.42 shows the comparison for selective regression testing (SR) vs. a full new regression test path suite (FR).

#### 4.3.4 RQ1: Applicability

We were successfully able to apply the regression testing approach. The changes were made to all types of artifacts (see above). Table 4.42 reports our results. It is organized the same as Table 4.40. Classifying behavioral tests due to changes to the Closing Documents cluster results in 4 retestable tests ( $|BT_r|$ ) and requires

Table 4.42: Selective Regression Testing vs. a Full New Regression Test Suite

	Phase #	Change BM		Change SE		Change F	
		SR	FR	SR	FR	SR	FR
1	$ BT_r $	4	0	1	0	12	0
	$ FMT_r $	0	0	0	0	0	0
	$ BT' $	4	0	0	0	0	0
2	Length $BT''$	106	3935	14	3998	169	3998
	Triplets (PE)	14	7976	1	7976	12	7962
	Reusable Triplets	26	0	0	0	0	0
3	$ FMT'' $	14	7978	1	7978	12	7964
	Length	174	96360	17	96359	68	96153
4	SR/FR (%)	< 1		< 1		< 1	
	Phase #	Change BM, SE, F		Change MM		Change WR	
		SR	FR	SR	FR	SR	FR
1	$ BT_r $	4	0	12	0	12	0
	$ FMT_r $	0	0	0	0	0	0
	$ BT' $	4	0	0	0	0	0
2	Length $BT''$	106	3935	169	3998	169	3998
	Triplets (PE)	23	7966	4	7972	3	7972
	Reusable Triplets	22	0	0	0	0	0
3	$ FMT'' $	23	7968	4	7974	3	7974
	Length	206	96206	66	96377	40	96377
4	SR/FR (%)	< 1		< 1		< 1	

4 new tests ( $|BT'|$ ). Their combined length is 106 nodes. Applying the coverage criteria generates 40 external failure mitigation test requirements, but 26 of these triplets are reusable, hence only 14 (35%) have to be converted into failure mitigation tests. This is a considerable saving. Only 14 external failure mitigation tests must be generated ( $|FMT''|$ ) at a length of 174. Selective regression testing requires only less than 1% of a full retest when the length is compared. The test effort in Boukhris et al. [70] was roughly proportional to test case length. While this cannot be assumed to hold for all systems, the large size of the web application may have led to averaging effects.



This means that one might be tempted to use this as an initial effort estimate using the same proportionality parameter as in [45], but due to the very small selective regression test suite, it would not be advisable to do that. We will explore this further in Subsection 4.3.8. Changes to the applicability matrix SE only require one new failure mitigation test. This is because the node where  $f_7$  has become applicable only occurs in one behavioral test. The mitigation (fix and proceed) adds 3 nodes to the test path length of the single reusable behavioral test (cf. Table 4.13).

Adding power outage as a new failure requires building a new SE Matrix for all 12 tests. The coverage criteria is fulfilled with 12 triplets as it covers all tests and all unique nodes.

When multiple changes to BM, SE, and F are made, the changes to the behavioral model determine the number of retestable and new tests (4 in each case), they are the same as for changes to BM only. Likewise, the length of the new test suite which is used to build the matrix for the potential failure scenarios is the same. The changes to applicability and the new row for failure  $f_{11}$  account for the higher number of required failure scenarios (28 instead of 22) and the reduction in reusable ones (22 as opposed to 24 due to changes in applicability).

Modifying the mitigation model for failure  $f_4$  makes all failure mitigation tests for failure  $f_4$  obsolete. We select all failure test requirement triplets (i, p, 4) in PE to build new failure mitigation tests. There are 4 such triplets, resulting in 4 new failure mitigation tests.

Similarly, changing the weaving rule for failure  $f_2$  makes all failure mitigation tests for failure  $f_2$  obsolete. We select all test requirement triplets (i, p, 2) in PE. There are three such triplets for which to build three new failure mitigation tests. Table 4.42 also reports on the savings in the regression test effort by removing

reusable test requirement triplets. Reusable triplets can occur whenever changes to a behavior model are made. The new approach to remove reusable triplets results in a 65% reduction when changes to the behavioral model are made and a 45% reduction when multiple changes are made (Rows 6/7 and 16/17 respectively). This is a substantial improvement.

Table 4.42 also allows a limited comparison of our selective regression testing approach to existing work: Andrews et al. [51] present the original FSMWeb approach but it did not discuss regression testing nor testing of fail-safe behavior. Table 4.42 shows in columns 2 and 3 under length BT” a comparison of selective regression testing for changes to the behavioral model using selective regression testing (106) vs a full retest using [51] (3935) for the behavioral model. Boukhris et al. [70] represent a full retest (columns marked FR); however, Table 31 reports data using coverage criteria, as opposed to the genetic algorithm used in Boukhris et al. [70]. Boukhris et al. [70] reports a total fail-safe test suite length of over 96,000 for the original model, given the limited amount of change, this is similar to the numbers reported here for a full retest. So, for this case study, having to test fail-safe behavior increases the test suite considerably. One might also want to consider test case length comparing Andrews et al. [51] and Andrews et al. [48], however, that is not possible due to the fact that the models are not the same, hence what they model is difficult to compare. Even a comparison between Andrews et al. [48] and Boukhris et al. [70] and this paper is limited, as this paper and Boukhris et al. [70] use 10 failure types and Andrews et al. [48] only uses 4. The same issues also limit the comparability of this paper and Andrews et al. [47].

### 4.3.5 RQ2: Efficiency of Selective Regression Testing

Rows 10 and 20 of Table 4.42 allow for a comparison between selective regression testing and a full retest. The savings are even more impressive than for the example (Table 4.40). This is because compared to the size of the behavioral model, the changes to the behavioral model are quite minor and localized to the CD subsystem.

For the behavioral model, we only added one node and two edges leading to 4 new test paths. This leads to a selective regression test suite which is only less than 1% of a full retest. We only make two applicability changes, only one of which affects potential failure scenarios. Less than 1% of a full retest is necessary. Deletion of  $f_1$  results in the removal of the failure with the most test requirement pairs. Adding failure  $f_{11}$  which is applicable in all states in CD is relatively expensive, but it is still much cheaper than a full retest. The most expensive selective regression test occurs with multiple changes, but is still only less than 1% of the test suite size of a full retest. As before in the example, changes to mitigations and weaving rules lead to much shorter selective regression tests compared to a full retest. Generally speaking, extensive changes to the behavior models are the most expensive. This is because it changes the behavioral test suite for selective regression testing the most. Since the behavioral test suite with the SE Matrix determines test requirement triplets, the test generation process in phases 2 and 3 is affected more, less of the existing failure mitigation tests are retestable or reusable. The later in the generation process that changes to artifacts occur, the more partial regeneration is possible, increasing efficiency.

In summary, unless a large web application experiences massive changes (and that is rare for mature software for stable application domains), we should expect selective regression testing to be a less expensive testing effort than a full retest.

### 4.3.6 RQ3: Genetic Algorithm vs. Coverage Criteria

In Boukhris et al. [70] a genetic algorithm was used to specify failure scenarios. This worked well because the search space (the potential failure scenarios SP) was rather large. Simulation experiments were conducted in Boukhris et al. [69] to compare the effectiveness of the genetic algorithm versus a number of coverage criteria. For the coverage criteria used here, the simulations indicate that (depending on the defect density of the external failure mitigation code) as the potential failure scenarios decrease, the coverage criteria outperformed the genetic algorithm. Most of the changes in our case study result in too few potential failure scenarios than a genetic algorithm can handle, hence it was necessary to use coverage criteria. A major reason for this is the removal of reusable potential failure scenarios which basically reduces the search space for the genetic algorithm. This is not only causing problems when using a genetic algorithm, it also increases efficiently dramatically when using coverage criteria.

### 4.3.7 RQ4: Reusable Failure Scenarios

One of the improvements in this paper is to remove reusable failure mitigation scenarios (triplets) from those that are required to be used in generating the failure mitigation test suite. Table 4.42 shows the number of triplets required (PE) and the number of triplets that are reused. Reusable triplets only occur when behavioral model changes are involved, with or without changes to failure applicability or failures.

For our case study, of 40 triplets selected with the coverage criteria only 14 are required, 22 are reusable. Only 35% of failure mitigation test requirements mandated by the coverage criteria are actually needed, meaning that only 35% of

tests need to be generated, executed and validated. When multiple changes are made the coverage criteria mandates 45 failure mitigation test requirements, of which 22 are reusable and can be removed, reducing PE to 51% of triplets required by the coverage criteria. Multiple changes increase the number of required triplets (e.g. when proper mitigation of a new external failure needs to be tested) and reduce reusable ones due to changes in failure applicability. Thus, the case study shows that identification and removal of reusable triplets are quite beneficial. Our case study shows significant efficiency improvement over Andrews et al. [47].

### 4.3.8 Threats to Validity

Wohlin et al. [227] define external validity as the extent to which it is possible to generalize the findings in a case study. As is common in case study research, external validity is an issue for ours as well. While we applied the regression testing method to a sizable web application (see Table 4.1) other web applications may not show the same positive results for improving the efficiency of regression testing. This depends partly on the types of changes. In our case, the changes were localized to a single component, enabling reuse of retestable tests. Had changes been made to every component (cluster model), all failure mitigation tests that are based on obsolete paths through these cluster models would have been obsolete resulting in far less efficiency than reported in Table 4.42. We thus do not claim that the results are typical. The running example in Section 4 had changes in both clusters, (FM1 and FM2), resulting in fewer savings through selective regression testing.

In Wohlin et al. [227], construct validity refers to the extent that the operational measures reflect what the researcher had in mind. There is a potential issue with construct validity, related to measuring the length of a test suite in terms of the

number of nodes. While in Boukhris et al. [70] length was roughly proportional to effort, we do not suggest this here for our case study. Given the small size of the selective regression test suite, the impact of a large number of inputs for some web pages vs. others might make the total number of inputs a better indicator of effort. This was also noticed in Andrews et al. [45] where tradeoff formulas comparing selective regression testing with brute force regression testing had two options as driving effort: length of test suite or size of inputs. In our case, Table 4.42 is trying to offer an efficiency comparison, This means that the actual efficiency improvement might be slightly less, but certainly will still be very, very good. Further, Andrews et al. [45] also report on an experiment with students to apply the selective regression testing approach in Andrews et al. [49]. They noticed that students in subsequent tasks became more efficient in applying the algorithms, indicating a learning effect. As they only worked with the graphs, not the inputs, the size of the inputs could not have made a difference. We, therefore, suggest that using the length of the test suite as part of an estimator of efficiency needs to be considered carefully in practice.

### 4.3.9 Practical Considerations

Because of the phased test generation, it is possible to use partial regeneration to replace obsolete test cases and thus, potentially improve efficiency. For example, an obsolete mitigation for failure  $e$ ,  $mt_e$ , only requires replacing a new  $mt'_e$  at the very last stage of generating failure mitigation tests, no changes are required for behavioral tests, generating required failure scenarios, nor weaving rules. We simply replace the obsolete mitigation paths with new ones.

Some changes require no new test generation, as when failure types no longer have to be mitigated or no longer apply in a particular state. This mainly re-

quires removing the obsolete models and tests, but no test execution. By contrast, widespread changes to the behavioral model may require regenerating most tests. If there are no changes to failures and their mitigation, we at least do not have to regenerate mitigation test paths; otherwise, we may as well start over. Andrews et al. [45] suggest to determine thresholds for changes to behavioral models. There are two thresholds, one for the proportion of FSM models changed and another for the proportion of edges changed to determine whether to use partial regeneration versus a full regeneration. Setting these thresholds initially depends on how conservative the estimate should be. A low threshold will lead to full regeneration, a higher one to more partial reconstruction. Similarly, when considering failure mitigation testing as we do, the threshold can be defined for a degree of change in mitigation models and applicability matrix. These changes tend to affect partial regeneration more than adding failures, for example, since that requires regeneration starting at the point of building a new failure applicability matrix through the remainder of the generation process. Still, the existing behavioral test suite can be reused as is.

If it is possible to localize changes and implement more extensive software changes in stages, the test generation for selective regression testing will be simpler, lead to fewer potential failure scenarios and fewer test cases. This was true for our case study, which only dealt with changes in one functional component, the Closing Documents Cluster.

In our approach, we assumed that the function of the software under test changed resulting in changes to models used in our MBT Black-box approach. What if defects were fixed, but no new functionality was added? In this case, there are no model changes. At the minimum, we have to rerun the failure mitigation tests that led to failures (i. e. uncovered the presence of defects). This may not be enough as changes to fix the defect may have introduced new defects. For example, when an incorrect

external failure mitigation for failure  $e$  is discovered one may want to rerun tests for all required failure scenarios for failure  $e$  again, even if they did not all uncover this mitigation defect. How far to go is a matter of risk trade-off, comparing the consequences of not testing external failure mitigation to the cost of testing their proper mitigation. This can involve prioritizing failure types and failure scenarios by the potential cost of a defective external failure mitigation. These priorities can then be used to run more important tests early until times runs out, accepting the consequences of defective external failure mitigation for less costly ones.



## Chapter 5

### Extensions: Testing Mobile Apps

Our approach to testing Mobile Applications using Finite State Machines (FSMApp) is an extension of FSMWeb [51] which is described in Subsection 3.1. Figure 5.1 shows the phases of the FSMApp process. FSMApp proceeds in three phases: Phase 1 builds a hierarchical model HFSM, Phase 2 generates tests from the HFSM, and Phase 3 compiles and executes tests through automated mobile testing tools.

#### 5.1 Testing Process for Mobile Apps

Functional testing for mobile app follows the following approach:

- Phase 1: Build a hierarchical model *HFSM*:
  - Partition the mobile app into clusters (*Cs*). (Subsection 5.3.1.)
  - Define Logical App Pages (*LAPs*) and Input-Action constraints for each page. (Subsection 5.3.2.)
  - Build FSMs for clusters as a multi-level hierarchy including an Aggregate FSM (*AFSM*) to represent the top level of the application. (Subsection 5.3.2.)

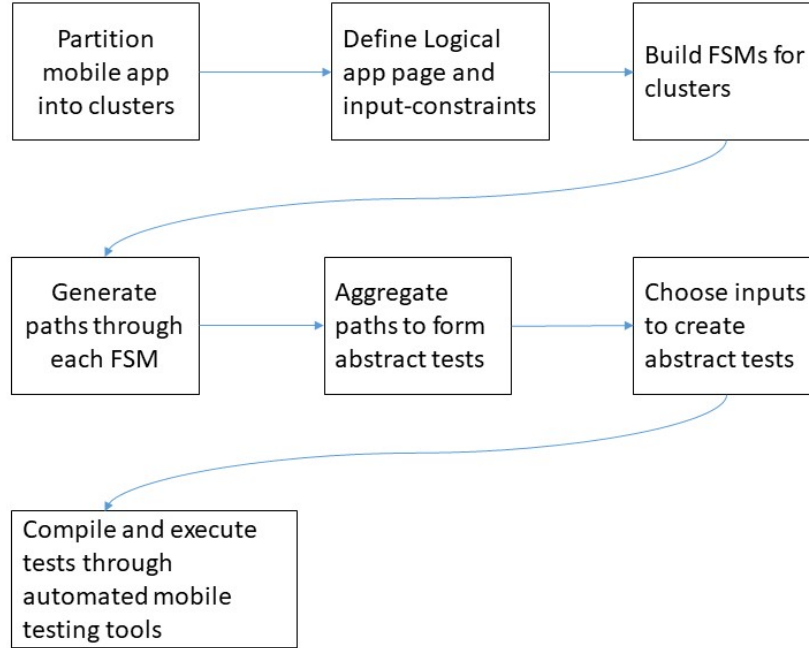


Figure 5.1: Process Mobile App

- Phase 2: Generate tests from the *HFSM*.
  - Generate paths through each FSM that meet the coverage criteria. (Subsection 5.4.1.)
  - Aggregate paths to form abstract tests. (Subsection 5.4.2.)
  - Choose inputs along the paths to create abstract tests. (Subsection 5.4.3.)
- Phase 3: Compile and execute tests through automated mobile testing tools. We will translate the abstract tests into a framework like Selenium and use Appium to execute the tests on a mobile device with the target mobile application. (Subsection 5.5.)

## 5.2 Example Used to Illustrate Approach

We illustrate our approach using the Family Medicines List application as the example. The Family Medicines List [9] is an open source mobile application offering the essential functions to manage medical information for a family. It is built using Basic4Android (B4A) [4] for Android operating systems. B4A is a graphical user interface tool to create native Android applications where the backend of the app is programmed in Java. Figure 5.2 shows screens for the Family Medicines List.

Appendix B shows a full set of screens and transitions for the App. The app has the following basic functionalities: (1) manage medicine information for a set of medicines. This includes adding, editing, deleting, and searching functions, (2) manage family member information alongside their medicine, (3) manage dosage and individual instruction for each medicine, and (4) list view with images for medicine lists. Figure 5.2 shows the three main pages of the Family Medicines List. Figure 5.2 (a) shows the list of all the medical information in the view list (component) of a family member after selecting a name from the list of users, Figure 5.2 (b) displays the form for user information management based on a database of all the information the user entered, and Figure 5.2 (c) add information about new medication page.

## 5.3 Phase 1: Build Model

### 5.3.1 Partition the Mobile App into Clusters (*Cs*).

The term *cluster* is used to refer to collections of software modules/app pages that implement a logical or user level function. The first step partitions the app application into clusters. At the highest level of abstraction, clusters represent

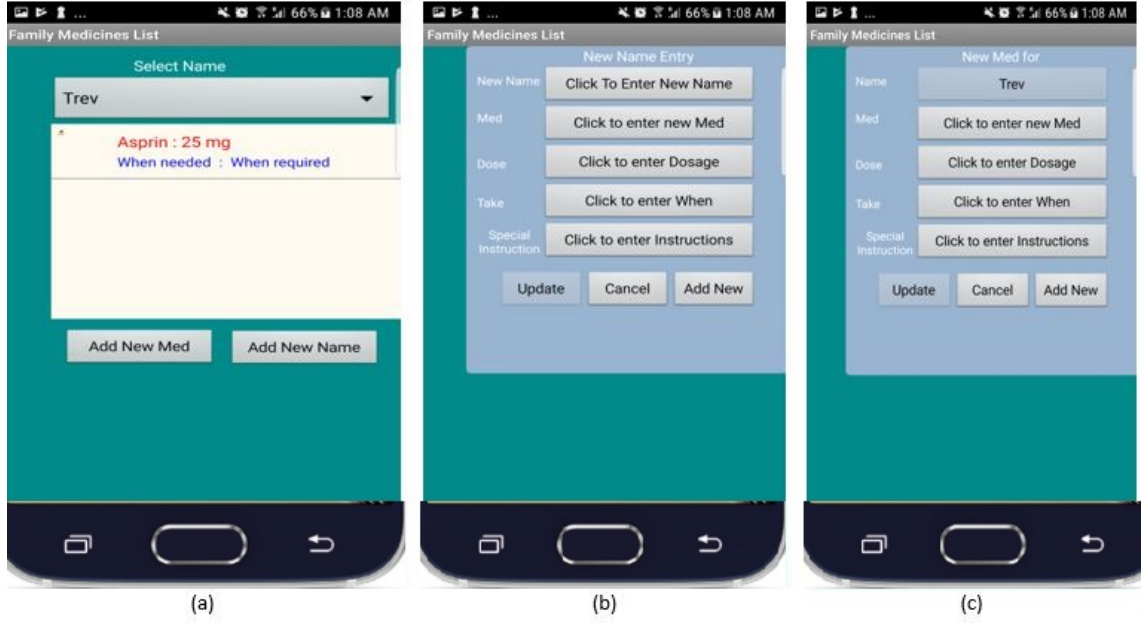


Figure 5.2: Family Medicines List App

functions that can be identified by users. Hence, an  $HFSM = \{FSM_i\}_{i=0}^n$  with a top level  $FSM_0 = AFSM$ . Each FSM has nodes that represent either Logical App Pages (LAPs) or clusters. Edges are internal or external to an FSM. External nodes span cluster boundaries. (They become internal at the next higher level.) External edges can either enter or leave a cluster FSM.

Clusters may be an individual Activity<sup>1</sup> or software modules that represent a major function. Clusters can be identified from the site navigation layout, coupling relationships among the components, and design information. Our example has one system to manage the medication of each member of a family. The lower level clusters are Info member medication (new name), Add and edit new medication of the family member. Figure 5.3 shows the top level of the Family Medicines List App. There are three main clusters: entering a new medication, entering a new name, modifying a medication and exiting the App.

<sup>1</sup>See Appendix A.

We will use Android mobile application to illustrate our approach. Android is a mobile operating system (OS) based on the Linux kernel [3]. It is designed for touch-screen mobile devices, for example, smartphones and tablets. Mobile applications fall in to three categories: native, web-based or hybrid. Native mobile applications are built to run directly on the OS. Web-based apps run on the browser of the device. A hybrid app is a combination of native app and web-based apps. Obviously since it is a black-box approach, it can be applied to other operating systems such as IOS or Windows with their components.

We apply FSMApp to the Family Medicines List App. Figure 5.3 shows the top level of the Family Medicines List App, while Figures 5.4 to 5.12 show the detail for the other clusters of the Family Medicines List App.

Appendix B presents Family medicines List App Screens. Screen (A) shows the main screen of the app. It has three functions: (1) Add new family member with new medication, dosage, when to take medicine and instructions. It presents as screen (C). (2) Add new medication for the available member of the family with all medication information (as shown in Screen (B) of Appendix B). (3) Edit and delete medication (as shown in Screen (C) of Appendix C).

We classify the main Screen (A) into three clusters (B, C, D) as shown in Figure 5.3. The AFSM for this example also the exit of the application as Exit LAP. We describe cluster (B), (C) and (D) in detail. Cluster (B) has four subclusters: New med in Figure 5.8, Dosage in Figure 5.9, When in Figure 5.10 and Instruction in Figure 5.11. We have two LAPs with two buttons, cancel and add New, as shown in Figure 5.5. Cluster (C) has five subclusters: "New name" in Figure 5.7, "New med" in Figure 5.8, "Dosage" in Figure 5.9, "When" in Figure 5.10 and "Instruction" in Figure 5.11. The Main Screen has two LAPs with two buttons, cancel and add New, as shown in Figure 5.4. Cluster (D) has one subcluster which is Edit in Figure 5.12.

We have two LAPs with two buttons, delete and cancel, as shown in Figure 5.6. Subcluster(E) has three subclusters: Dosage in Figure 5.9, When in Figure 5.10 and Instruction in Figure 5.11. We have two LAPs with two buttons, cancel and add New, as shown in Figure 5.12. Also, We have two LAPs with two buttons, cancel and accept, for each subclusters (F, G, H, I, J) shown in figs. 5.7 to 5.11 <sup>2</sup>.

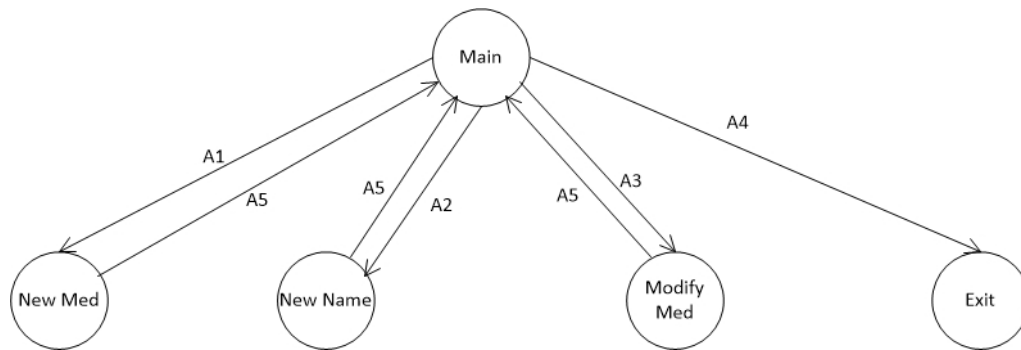


Figure 5.3: Main Page

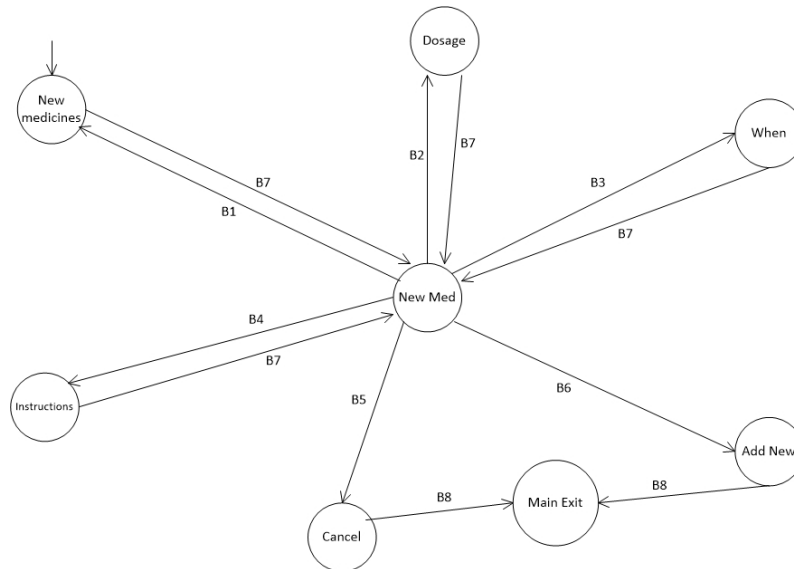


Figure 5.4: Family Medicines List New Med

---

<sup>2</sup>Appendix D.1 shows more details about the clusters nodes.

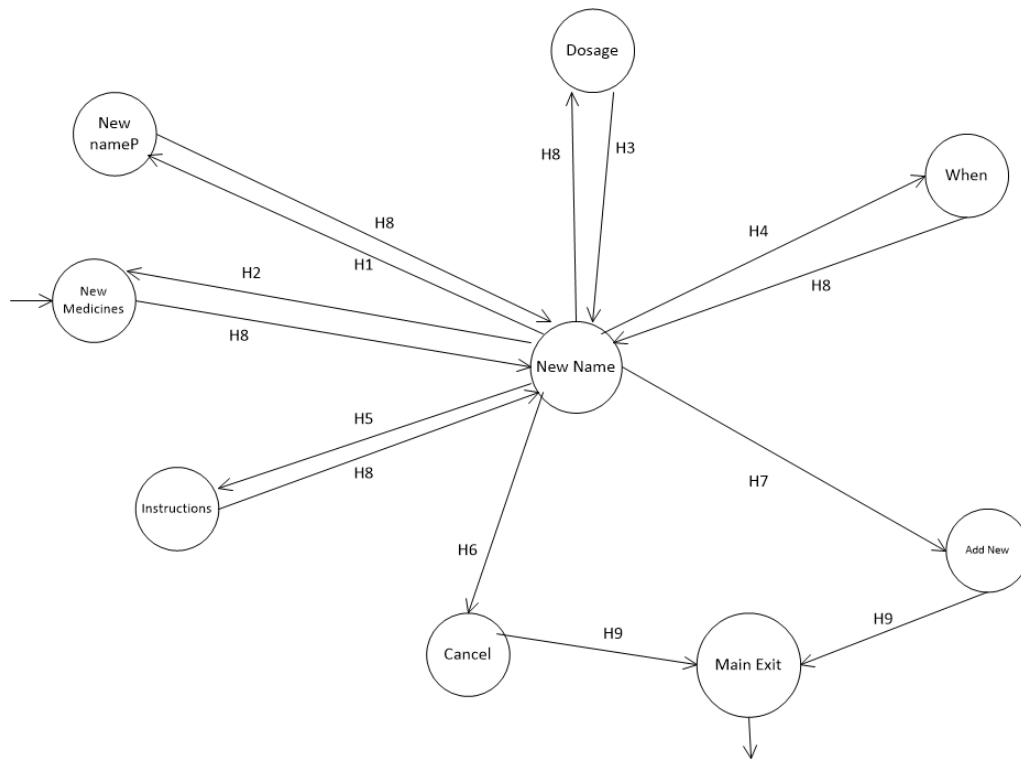


Figure 5.5: Family Medicines List New Name

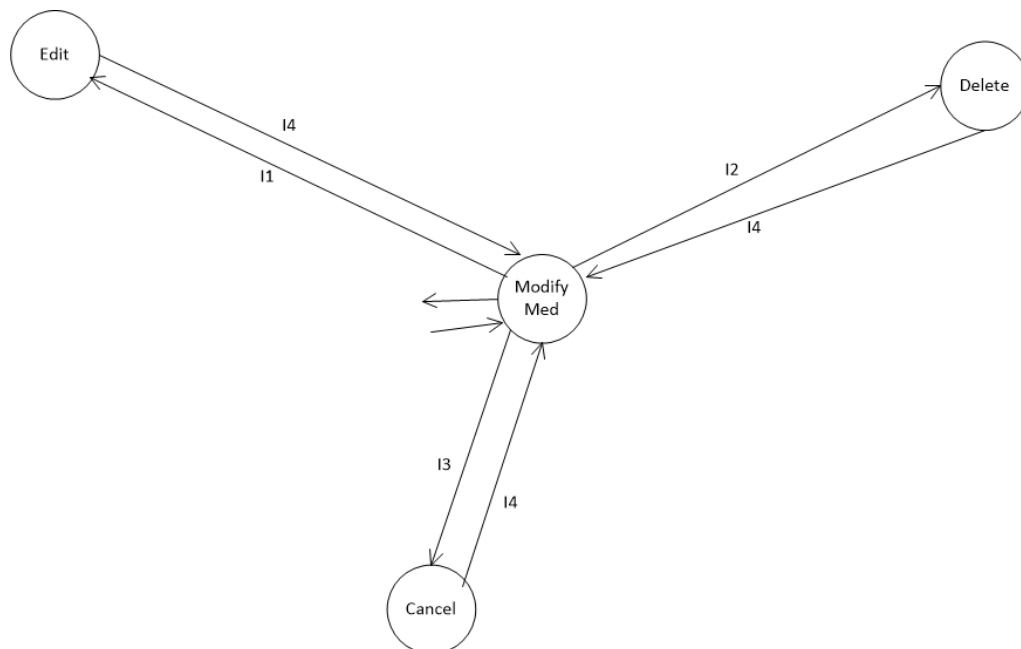


Figure 5.6: Family Medicines List Modify Med

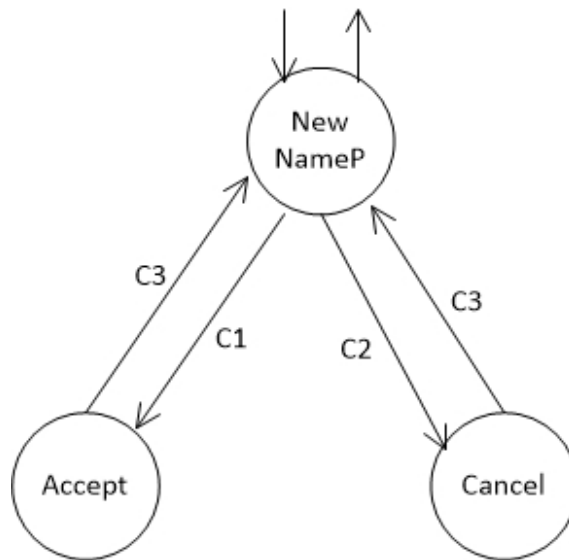


Figure 5.7: Family Medicines List New NameP

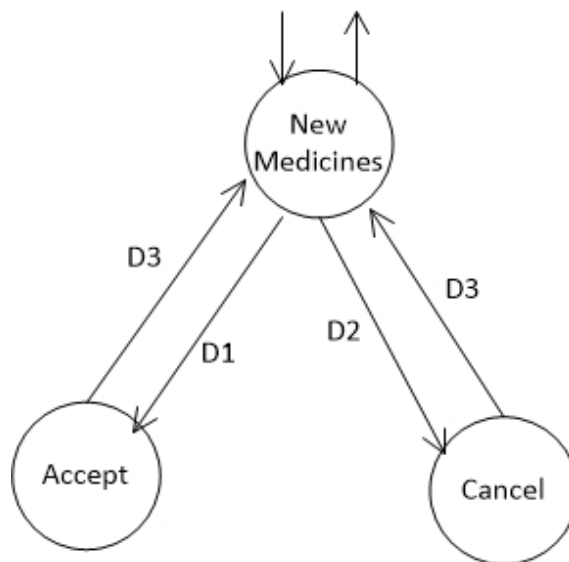


Figure 5.8: Family Medicines List New Medicines



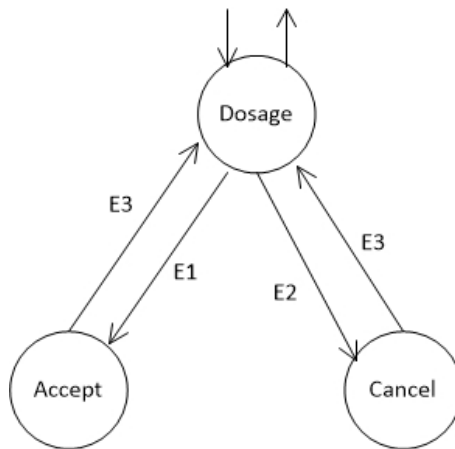


Figure 5.9: Family Medicines List Dosage

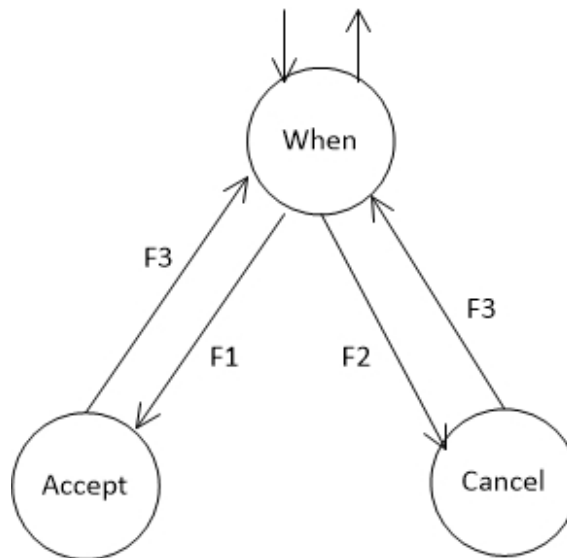


Figure 5.10: Family Medicines List When

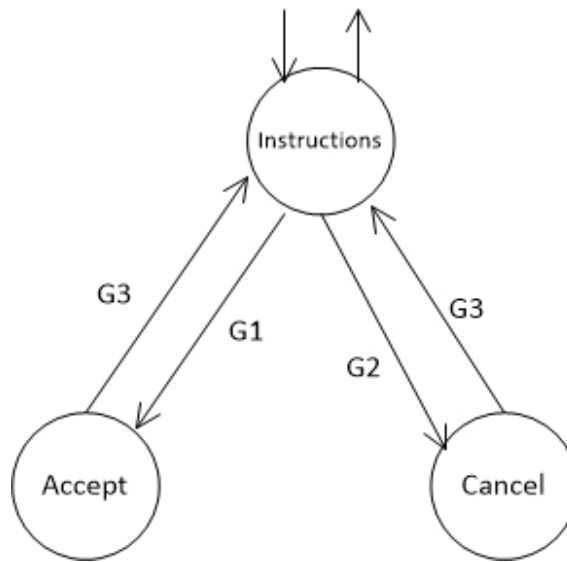


Figure 5.11: Family Medicines List Instructions

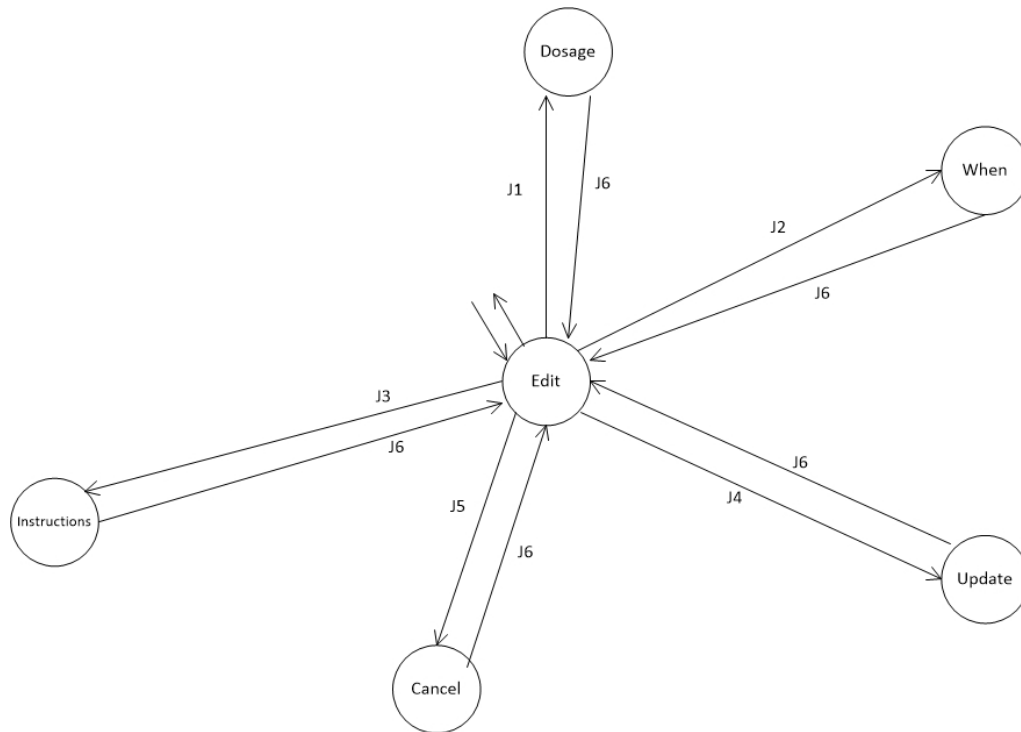


Figure 5.12: Family Medicines List Edit

### 5.3.2 Define Logical App Pages (*LAPs*) and Input-Action Constraints for Each

Mobile apps have a variety of screens. We will consider screens as input components, or logical app pages (LAPs), and the inputs and their constraints on these LAPs next. For illustration purposes, we list the actives (screens) for Android. Similar components exist for other types of mobile devices. Many app actives (screens) contain XML forms, each of which can be connected to a different back-end software module. To facilitate testing of these modules, app pages are modeled as multiple *Logical App Pages* (LAPs). A LAP is either a physical app page, physical app component, or the portion of an app activity that accepts data from the user through a XML form, and then sends the data to a specific software module. LAPs are abstracted from the presentation defined by the XML and are described in terms of their sets of *inputs* and *actions*. FSMApp is an MBT meant for black-box testing hence, the mobile application can be written in any language appropriate for mobile applications (e.g. Ruby, JavaScript, HTML, etc.). All inputs in a LAP are considered atomic: data entered into a text field is considered to be only one user input symbol, regardless of how many characters are entered into the field.

There may be rules about the inputs: some inputs may be required, while others may be optional; users may be allowed to enter inputs in any order, or a specific order may be required. Table 5.1 shows the input constraints, and the order of the inputs. Required (R) means that required input must be entered. Required Value (R(parm)) means that one must enter at least one value. Optional (O) means that an input may or may not be entered. Single Choice (C1) means that one input should be selected from a set of choices, and Multiple choice means that more than one input should be selected from a set of choices. Table 5.2 shows how typical

input types found in mobile applications are represented as constraints on edges in an FSMApp model. The difference between the web input types and mobile input types are swipe and scroll. Swipe (W) means that a swipe is required to change the value of a component. Scroll (L) indicates that the input required is to scroll up or down the content. Our language express the input constraints in a BNF grammar in Appendix C.

Input Choice	Order
Required (R)	Sequence (S)
Required Value (R(parm))	Any (A)
Optional (O)	
Single choice (C1)	
Multiple choice (Cn)	

Table 5.1: Constraints on Inputs

FSMWeb [51] and FSMApp differ in their input types. FSMApp has many more input types than FSMWeb. FSMApp models components, whereas FSMWeb does not provide for components. In our Family Medicine Example, Figure 5.2 shows the medicine list as a lists component.

Mobile applications can have a variety of components that can be modeled via input constraints. While they vary a little between different mobile operating systems, they also have many types of components in common. To illustrate what components look like we explain common components of Android applications and what FSMApp’s input constraints would look like <sup>3</sup>:

---

<sup>3</sup>These are the 21 components that Google lists for Android Apps.

1. A **bottom sheet** 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  $R(<Click>)$ .
2. A **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  $C1(\text{Select Button}, \text{Click})$ .
3. A **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  $C1(<Click>, <Swipe>, <Scroll>, <Pick>)$ .
4. **Chips** represent complex entities in small blocks, such as a contact. The action of the chip is a click. The input constraint is  $R(S(<Select Button>, <Click>))$ .
5. **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 text editing. The input constraint is  $C1(<Row\ hover>, <Selection>, <Sort>, <Column\ hover>, <Edit>)$ .
6. **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  $R(<Click>)$ .
7. **Dividers** group and separate content within lists and page layouts. The action of the divider is click. The input constraint is  $R(<Click>)$ .

8. **Grid lists** are an alternative to standard list views. The actions of the grid list are vertical scrolling or filtering. The input constraint is  $C1(<Scroll>, <Filter>)$ .
9. **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  $R(<Sort>)$ .
10. **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  $C1(<Scroll>, <Click>)$ .
11. **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 pickers are drop-down and click. The input constraint is  $C1(<Dropdown>, <Click>)$ .
12. **Progress & activity** indicators are visual indications of an app loading content. The action of Progress & Activity is loading. The input constraint is  $R(<Load>)$ .
13. **Selection Controls** allow the user to select options. The action of selection controls is click. The input constraint is  $R(<Click>)$ .
14. **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  $R(<Scroll>)$ .
15. **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  $R(<Click>)$ .

16. **Subheaders** 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>).
17. **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>).
18. **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 tab is scroll. The input constraint is R(<Scroll>).
19. **Toolbars** appear above the view affected by their actions. The action of toolbars is scroll. The input constraint is R(<Scroll>).
20. **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>).
21. **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 lookup table, select text, and write text. The input constraint is C1(<Lookup>, <Select>, <Write>).

Table 5.2 summarizes the components for Android Apps. LAPs are at the lowest model level. They can be an input type as defined in Table 5.2 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 5.2 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 5.2: 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	A. R(Image, click)  B. R(Image, click) C. R(Video, click) D. R(Textbox) E. R(Text Area)	Display  Change size Run Display text Display many lines of text	Text
Continued on next page					



**Table 5.2 – continued from previous page**

No	Components	Interface Controls	Actions	Effect	Input Type
		F. Links	F. R(Button, click) G. R(Link, click)	Show other website or page	
4 (c)	Chips	A. Textbox  B. Cards	A. R(Enter text)  B. R(Display content, choose) C. R(Click on chip)	Save  Show details  Display card	Text
5 (c)	Data Tables	A. Checkbox  B. Link  C. Textbox D. Menu E. Button F. Card	A. R(Select, select dialog, add content) B. R(Select, click button) C. R(Click link) D. R(Select)	Save  Delete  Transfer Show card	Text  Non-Text
6 (c)	Dialogs	A. Button  B. Textbox  C. Date picker	A. R(Show warning, click close) B. R(Select dialog, input content) C. R(Select dialog, click date picker, choose date, close)	 Save  Save	Text  Non-Text
Continued on next page					

**Table 5.2 – continued from previous page**

No	Components	Interface Controls	Actions	Effect	Input Type
		D. Checkbox  E. Time picker  F. Radio Box G. Menu H. Bar slide input	D. R(Select dialog, click Time picker, choose date, close) E. R(Click menu, choose from list)	Save  Close	
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	A. R(Select text, copy) B. R(Select combobox, choose content)	Show list in a menus Links of button to another pages	Text Non-Text
Continued on next page					

**Table 5.2 – continued from previous page**

No	Components	Interface Controls	Actions	Effect	Input Type
		C. Combobox  D. Checkbox E. Switch F. Reorder G. Ex- pand/collapse H. Leave-behinds	C. R(Select text, write content)		
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	
Continued on next page					

**Table 5.2 – continued from previous page**

No	Components	Interface Controls	Actions	Effect	Input Type
				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
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 feed-back 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
Continued on next page					

**Table 5.2 – continued from previous page**

No	Components	Interface Controls	Actions	Effect	Input Type
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 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	A. R(Content, click)	Save the content	Text

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

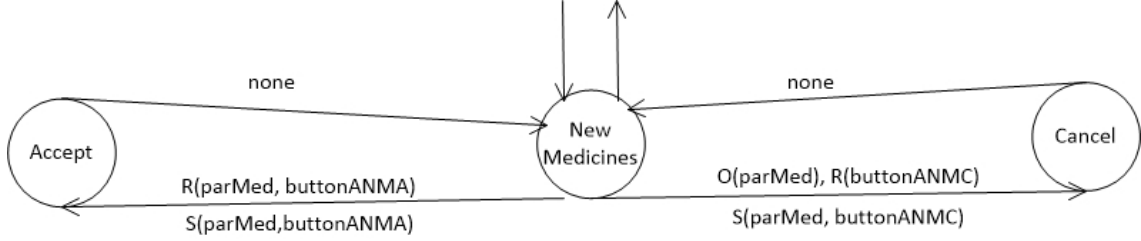


Figure 5.13: Annotated FSM for New Medicines Cluster of Table 5.8

or cluster. Figure 5.13 shows the Add New Medicines input-action constraints. In the New Medicines cluster, there are two states: either you enter the new medicine by name (parMed) and accept it (buttonANMA), or you cancel the new medicine (with or without giving the medicine name). 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.

Tables 5.3 to 5.12 show the transitions, explanation, and input action constraints. Column 1 uniquely 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 5.3 shows 5 transitions for the main page cluster (AFSM). The transitions connect the main page with four clusters and one LAP (Exit App). Table 5.4 shows 8 transitions for New Med cluster to connect with 4 clusters and two LAP nodes (Update and Cancel). Table 5.5 shows 9 transitions for New New cluster to connect with 5 clusters and two LAP nodes (Update and Cancel). Table 5.6 shows 4 transitions for modify med cluster to connect with Edit medicine cluster, delete LAP and cancel LAP. Table 5.7 shows 3 transitions for New NameP cluster to connect with the accept LAP and the cancel LAP. Table 5.8 shows 3 transitions for the New Medicines cluster to connect with the accept LAP and the cancel LAP. Table 5.9 shows 3 transitions for the Dosage cluster to connect with the accept LAP and the cancel

LAP. Table 5.10 shows 3 transitions for the When cluster to connect with the accept LAP and the cancel LAP. Table 5.11 shows 3 transitions for the Instructions cluster to connect with the accept LAP and the cancel LAP. Table 5.12 shows 6 transitions for the Edit cluster to connect with 3 clusters (Dosage, When, Instructions) and two LAP nodes (Update and Cancel).

Table 5.3: Transitions of Figure 5.3 (Main Page Cluster AFSM)

Transition	Explanation	Constraints
A1	Access New Medicines	R(selectN, buttonANM) S(selectN, buttonANM) Continue-use(SelectN)
A2	Access New Name	R(buttonANN)
A3	Access Modify Medicines	O(selectN), R(selectM) S(selectN, selectM) Continue-use(SelectN, SelectM)
A4	Exit the System	R(buttonBack)
A5	Back to Main Page	none

Table 5.4: Transitions of Figure 5.4 (New Med Cluster)

Transition	Explanation	Constraints
B1	Access New Medicines	R(buttonANM)
B2	Access Dosage	R(buttonAD)
B3	Access When	R(buttonAW)
B4	Access Instructions	R(buttonAI)
B5	Cancel Add New Med	R(buttonCNM)
B6	Add New Medicines	R(buttonAM)
B7	cancel to New Med	none
B8	Back to Main Page	none

Table 5.5: Transitions of Figure 5.5 (New Name Cluster)

Transition	Explanation	Constraints
H1	Access New NameP	R(buttonANNP)
H2	Access New Medicines	R(buttonANM)
H3	Access Dosage	R(buttonAD)
H4	Access When	R(buttonAW)
H5	Access Instructions	R(buttonAI)
H6	Cancel Add New Name	R(buttonCNN)
H7	Add New Name	R(buttonANNA)
H8	cancel to New Name	none
H9	Back to Main Page	none



Table 5.6: Transitions of Figure 5.6 (Modify Med Cluster)

Transition	Explanation	Constraints
I1	Edit the Medicine	O(Parname, parMed), R(buttonE) S(Parname, parMed, buttonE) Continue-use(parName, parMed)
I2	Delete the Medicine	O(Parname, parMed), R(buttonD) S(Parname, parMed, buttonD)
I3	Cancel to Previous Page	O(Parname, parMed), R(buttonCE) S(Parname, parMed, buttonCE)
I4	Back to Previous Page	none

Table 5.7: Transitions of Figure 5.7 (New NameP Cluster)

Transition	Explanation	Constraints
C1	Add the new name	R(parName, buttonANNPA) S(parName, buttonANNPA) Continue-use(parName)
C2	Cancel to Previous Page	O(parName), R(buttonANNPC) S(parName, buttonANNPC)
C3	Back to Previous Page	none

Table 5.8: Transitions of Figure 5.8 (New Medicines Cluster)

Transition	Explanation	Constraints
D1	Add the new Med	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)
D2	Cancel to Previous Page	O(parMed), R(buttonANMC) S(parMed, buttonANMA)
D3	Back to Previous Page	none

Table 5.9: Transitions of Figure 5.9 (Dosage Cluster)

Transition	Explanation	Constraints
E1	Add the new dosage	R(parDosage, buttonADA) S(parDosage, buttonADA) Continue-use(parDosage)
E2	Cancel to Previous Page	O(parDosage), R(buttonADC) S(parDosage, buttonADC)
E3	Back to Previous Page	none

Table 5.10: Transitions of Figure 5.10 (When Cluster)

Transition	Explanation	Constraints
F1	Add the new when	R(parWhen, buttonAWA) S(parWhen, buttonAWA) Continue-use(parWhen)
F2	Cancel to Previous Page	O(parWhen), R(buttonAWC) S(parWhen, buttonAWC)
F3	Back to Previous Page	none

Table 5.11: Transitions of Figure 5.11 (Instructions Cluster)

Transition	Explanation	Constraints
G1	Add the new instructions	R(parInstructions, buttonAIA) S(parInstructions, buttonAIA) Continue-use(parInstructions)
G2	Cancel to Previous Page	O(parInstructions), R(buttonAIC) S(parInstructions, buttonAIC)
G3	Back to Previous Page	none

Table 5.12: Transitions for Figure 5.12 (Edit Cluster)

Transition	Explanation	Constraints
J1	Access Dosage	R(buttonAD)
J2	Access When	R(buttonAW)
J3	Access Instructions	R(buttonAI)
J4	Update the Medicine	R(buttonEEU)
J5	Cancel to Previous Page	R(buttonEEC)
J6	Back to Previous Page	none

In addition to input-action constraints, there may be rules concerning how and whether selected input values may be or must be reused (propagated). The types of the propagated input values are:

- Continue-use: the selected input values must be reused later in the test path. For example, the patient name must be passed to add new medicines cluster.
- Single-use: the selected input value must be used only once. For example, when one deletes the medicine of a patient, it cannot be used in the test again, unless a new medicine is added with this name.
- Not-propagated: The input has no constraints on reuse. We may or may not use it later in the test.

Table 5.13 shows the input constraints and the propagation rules for the main page (AFSM) of Figure 5.3. Column 1 shows the Transition Annotation. Column 2 shows the input constraints.

Table 5.13: Annotations for Main Page (AFSM) Transitions of Figure 5.3

Transition	Constraints
A1	O(selectN), R(selectM) S(selectN, buttonANM) continue-use(selectN)
A2	R(buttonANN)
A3	O(selectN), R(selectM) S(selectN, selectM) continue-use(selectN, selectM)
A4	R(buttonBack)
A5	none

## 5.4 Phase 2: Generate Test Sequences

### 5.4.1 Paths through FSMs/AFSM

Test sequences are generated during phase 2 of the FSMApp method. The user can select coverage criteria such as node, edge, edge-pair, simple round trip and prime path coverage. A test sequence is a sequence of transitions through the aggregate 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*.

We generate the test sequences for each cluster that satisfy edge coverage. Tables 5.14 to 5.23 show test paths for each cluster as sequences of nodes. The corresponding graphs are mentioned in the caption. Nodes in bold indicate the node is a cluster node. For this App, with 10 clusters, several clusters only need a single test path and only one cluster needs 4 paths. The total number of paths is 20. The paths are relatively short.

Table 5.14: Main Page Test Sequences of Figure 5.3

ID	Test Sequence	Length
1	[Main, <b>New Med</b> , Main, Exit]	4
2	[Main, <b>New Name</b> , Main, Exit]	4
3	[Main, <b>Modify Med</b> , Main, Exit]	4

Table 5.15: New Med Cluster Test Sequences of Figure 5.4

ID	Test Sequence	Length
1	[ <b>New medicines</b> , New Med, <b>New medicines</b> , New Med, <b>Dosage</b> , New Med, Cancel, Main Exit]	8
2	[ <b>New medicines</b> , New Med, <b>When</b> , New Med, Cancel, Main Exit]	6
3	[ <b>New medicines</b> , New Med, <b>Instructions</b> , New Med, Add New, Main Exit]	6

Table 5.16: New Name Cluster Test Sequence of Figure 5.5

ID	Test Sequence	Length
1	[ <b>New medicines</b> , New Name, <b>New medicines</b> , New Name, <b>New nameP</b> , New Name, Cancel, Main Exit]	8
2	[ <b>New medicine</b> , New Name, <b>Dosage</b> , New Name, Cancel, Main Exit]	6
3	[ <b>New medicines</b> , New Name, <b>When</b> , New Name, Cancel, Main Exit]	6
4	[ <b>New medicines</b> , New Name, <b>Instructions</b> , New Name, Cancel, Main Exit]	6

Table 5.17: New NameP Cluster Test Sequences of Figure 5.7

ID	Test Sequence	Length
1	[New NameP, Accept, New NameP, Cancel, New NameP]	5

Table 5.18: New Medicines Cluster Test Sequences of Figure 5.8

ID	Test Sequence	Length
1	[New medicines, Accept, New medicines, Cancel, New medicines]	5

Table 5.19: Dosage Cluster Test Sequences of Figure 5.9

ID	Test Sequence	Length
1	[Dosage, Accept, Dosage, Cancel, Dosage]	5

Table 5.20: When Cluster Test Sequences of Figure 5.10

ID	Test Sequence	Length
1	[When, Accept, When, Cancel, When]	5

Table 5.21: Modify Med Cluster Test Sequences of Figure 5.6

ID	Test Sequence	Length
1	[Modify Med, <b>Edit</b> , Modify Med]	3
2	[Modify Med, Cancel, Modify Med, Delete, Modify Med]	5

Table 5.22: Instructions Cluster Test Sequences of Figure 5.11

ID	Test Sequence	Length
1	[Instructions, Accept, Instructions, Cancel, Instructions]	5

Table 5.23: Edit Cluster Test Sequences of Figure 5.12

ID	Test Sequence	Length
1	[Edit, <b>Dosage</b> , Edit, Cancel, Edit]	5
2	[Edit, <b>When</b> , Edit, Cancel, Edit]	5
3	[Edit, <b>Instructions</b> , Edit, Cancel, Edit]	5

By the end of this step, we generated all the test paths for each cluster.

### 5.4.2 Path Aggregation

The test sequences through FSM in HFSM are now aggregated into test sequences for the whole model. A number of aggregation criteria have been proposed: all-combinations, each choice and base choice coverage [42]. We apply all-combinations coverage. This is the most expensive aggregation coverage criteria. The process results in a set of aggregate paths. We call them *abstract tests*. Algorithm 1 shows the procedure to aggregate test paths .

The inputs to the algorithm are AFSM and cluster test paths. The output of algorithm 1 is a set of aggregated test paths. Line 1 of the algorithm copies AFSM test paths into an input List. Line 2 iterates through every test path from the input list. Line 3 takes one test path from the list. Then, we sequentially check 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.

For example, the first test path in AFSM [Main, **New Med**, Main, Exit] in Table 5.14 can be aggregated as follows: The test path has one cluster node (New med). The cluster node should be replaced by the New Med cluster test paths from



**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  $node_i$  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

Table 5.15. The results of this step are three paths:

1. Main, **New Medicines**, New Med, **New Medicines**, New Med, **Dosage**, New Med, Cancel, Main Exit, Main, Exit
2. Main, **New Medicines**, New Med, **When**, New Med, Cancel, Main Exit, Main, Exit
3. Main, **New Medicines**, New Med, **Instructions**, New Med, Add New, Main Exit, Main, Exit

Test 1 still has three cluster nodes: New Medicines, New Medicines and Dosage cluster nodes. Test 2 has New Medicines and When cluster nodes. Test 3 has New Medicines and Instructions cluster nodes. After we replace all the cluster nodes, we get the following test paths.

1. Main, New medicines, Accept, New medicines, Cancel, New medicines, New Med, New medicines, Accept, New medicines, Cancel, New medicines, New Med, Dosage, Accept, Dosage, Cancel, Dosage, New Med, Cancel, Main Exit, Main, Exit
2. Main, New medicines, Accept, New medicines, Cancel, New medicines, New Med, When, Accept, When, Cancel, When, New Med, Cancel, Main Exit, Main, Exit
3. Main, New medicines, Accept, New medicines, Cancel, New medicines, New Med, Instructions, Accept, Instructions, Cancel, Instructions, New Med, Add New, Main Exit, Main, Exit

Table 5.24 shows the aggregated test paths of the Family Medicines list app as sequences of nodes. Column one shows the id of the test path. Column two shows

the abstract test paths. Column three shows the length of the test paths in terms of nodes. The total length of the test paths is 181 nodes. The longest path consists of 23 nodes, but there are only two of those. The shortest has 8 nodes. Median length is 17 nodes.

Id	Test Path	Length
1	[Main, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Med, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Med, Dosage, <i>Accept</i> , Dosage, <i>Cancel</i> , Dosage, New Med, <i>Cancel</i> , <i>Main Exit</i> , Main, <i>Exit</i> ]	23
2	[Main, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Med, When, <i>Accept</i> , When, <i>Cancel</i> , <i>When</i> , New Med, <i>Cancel</i> , <i>Main Exit</i> , Main, <i>Exit</i> ]	17
3	[Main, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Med, Instructions, <i>Accept</i> , Instructions, <i>Cancel</i> , <i>Instructions</i> , New Med, <i>Add New</i> , <i>Main Exit</i> , Main, <i>Exit</i> ]	17
4	[Main, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Name, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Name, New NameP, <i>Accept</i> , New NameP, <i>Cancel</i> , <i>New NameP</i> , New Name, <i>Cancel</i> , <i>Main Exit</i> , Main, <i>Exit</i> ]	23
5	[Main, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Name, Dosage, <i>Accept</i> , Dosage, <i>Cancel</i> , <i>Dosage</i> , New Name, <i>Cancel</i> , <i>Main Exit</i> , Main, <i>Exit</i> ]	17
6	[Main, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Name, When, <i>Accept</i> , When, <i>Cancel</i> , <i>When</i> , New Name, <i>Cancel</i> , <i>Main Exit</i> , Main, <i>Exit</i> ]	17
7	[Main, New medicines, <i>Accept</i> , New medicines, <i>Cancel</i> , <i>New medicines</i> , New Name, Instructions, <i>Accept</i> , Instructions, <i>Cancel</i> , <i>Instructions</i> , New Name, <i>Add New</i> , <i>Main Exit</i> , Main, <i>Exit</i> ]	17
8	[Main, Modify Med, <i>Cancel</i> , Modify Med, <i>Delete</i> , <i>Modify Med</i> , Main, <i>Exit</i> ]	8

9	[Main, Modify Med, Edit, Dosage, <i>Accept</i> , Dosage, <i>Cancel</i> , <i>Dosage</i> , Edit, <i>Cancel</i> , <i>Edit</i> , <i>Modify Med</i> , Main, <i>Exit</i> ]	14
10	[Main, Modify Med, Edit, When, <i>Accept</i> , When, <i>Cancel</i> , <i>When</i> , Edit, <i>Cancel</i> , <i>Edit</i> , <i>Modify Med</i> , Main, <i>Exit</i> ]	14
11	[Main, Modify Med, Edit, Instructions, <i>Accept</i> , Instructions, <i>Cancel</i> , <i>Instructions</i> , Edit, <i>Cancel</i> , <i>Edit</i> , <i>Modify Med</i> , Main, <i>Exit</i> ]	14
	Total Length	181

Table 5.24: Aggregated Test Paths

When we build the model, we added dummy nodes and transitions to ensure single-entry-single exit cluster models. Table 5.24 shows them in italics. These do not require any inputs as they are not really testing steps. However, they increase the length of the test paths. The next step removes these dummy nodes and transitions and replaces each remaining node pair (edge) with its corresponding input action constraint.

**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  $\text{Length}(\text{path}_i) - 1$  do
3:     if  $\text{edge}(\text{node}_j, \text{node}_{j+1})$  has constraint then
4:       add to constraint sequence for  $\text{path}_i$ 
5:     end if
6:   end for
7: end for
```

**Algorithm 2:** Test Step Reduction

Algorithm 2 shows the procedure for test step reduction. The algorithm has two inputs: the set of Aggregated Paths and the number of paths. The output of the algorithm is the sequence of input constraints for each aggregated test path. The algorithm has two loops: 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. Table 5.25 shows the result of the test step reduction for the first aggregated test path. Column one shows the id of the test path. Column two shows the Transition Id for constraint sequence. Column three shows the Constraint Sequence of the reduced test path. Each input action constraint is separated by a horizontal line. The last column shows the length of the constraint sequence.

Note that test path length is reduced by more than half, from 23 to 11 steps. We added the input constraints based on Tables 5.3 to 5.12. Table D.11 in Appendix D shows the aggregated test paths with the transitions. The first test path of Table D.11 can be reduced by removing the transitions without constraints. The transitions without constraint are A5, B7, B8, D3, and E3. Table D.11 in Appendix D shows the test sequences in the form of input constraint sequences for all aggregated paths in its column 3.

One of the goals of this approach is to extend the FSMWeb to test mobile applications while keeping the size and the complexity of test paths manageable. For this example, there are 11 test sequences, their length varying between 4 and 11 with a total length of 87. Table 5.26 shows length before and after the reduction step. Column 1 shows the id of the test path. Column 2 shows the length of the aggregated test path from Table 5.24 in terms of number of nodes. Column 3 shows the length of the test paths after the reduction as number of edges in order to compare the result with the other approaches in Chapter 6. Column 4 and 5 show the inputs and actions. Since, every step includes one action, the number of actions and test steps is the same. Input action constraints may or may not require inputs or multiple inputs before an action (like a button click) leads to a transition event. We will discuss Column 4 and Column 5 in more detail in subsection 5.4.3. The last row of the table shows the total length of aggregated test paths as 181 nodes

Table 5.25: Test Path After Reduction Step

ID	Edge Id	Constraint	Length
1	A1	R(SelectN, buttonANM) S(SelectN, buttonANM) continue-use(SelectN)	11
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	
	A1	R(buttonANM)	
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parName)	
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	
	H3	R(buttonAD)	
	E1	R(parDosage, buttonADA) S(parDosage, buttonADA) Continue-use(parDosage)	
	E2	O(parDosage), R(buttonADC) S(parDosage, buttonADC)	
	B6	R(buttonAM)	
	A4	R(buttonBack)	

Table 5.26: Length of Before and After Reduction Step

Test Path ID	Before Reduction Step	After Reduction Step	Inputs	Actions
1	23	11	6	11
2	17	8	4	8
3	17	8	4	8
4	23	11	6	11
5	17	8	4	8
6	17	8	4	8
7	17	8	4	8
8	8	4	4	4
9	14	7	4	7
10	14	7	4	7
11	14	7	4	7
Total	181	87	48	87

(170 edges) and the total length of the test sequence after the reduction step as 87 transitions. We reduce the transitions by 49%. After the reduction step, the longest test sequence consists of 11 transitions, but there are only two of those. The shortest has 4 transitions. The median length is 8 transitions. The reduction step helps to keep the size and complexity of the test sequences manageable.

### 5.4.3 Input Selection

The final step of test generation is selecting inputs to replace the input constraints in the test sequence constructed in subsection 5.4.2. The test values are selected by the test designer.

We do not require specific input domain coverage to keep the test designer free to make their own decision in this regard. For example, the test designer can generate input values by covering partitions or randomly selecting values from a list as long as input selection constraints are met. The test designer chooses values for related inputs. For example, the test designer chooses which medicine to match

with the patient name. At the end of this step, we have a set of inputs for the execution phase. Table 5.27 shows the set of inputs for test path 8 of Table 5.24. Column one shows the constraint sequence, and column two shows the input values that meet the constraints. The last column explains each value. Test 8 has four inputs and five actions. The inputs are patient name (parname) and medicine name (parMed) which occurs twice in test 8. The input values are selected by the test designer. Both ad-hoc and coverage based value selection are possible. The actions are select patient name (SelectN), select medicine (SelectM), Click delete button (buttonD), click cancel button (buttonCE) and click back arrow to exit the mobile app (buttonBack).

In Appendix D, Table D.11 shows the input selection for all test sequences of the Family Medicine List app. Table D.11 is structured similarly to Table 5.27, except that it adds Test Id as the first column. The total number of inputs is 48, and the total number of actions is 87. There is a difference between the inputs and the actions because some of the test steps only require clicks (actions). The unique inputs are name (parname), medicine name (parMed), amount of dosage (parDosage), time of medicine (parWhen) and medicine instruction (parInstruction). There are five unique inputs, and 27 unique actions (Buttons or Combobox).

## 5.5 Phase 3: Execute and Validate Tests

Unlike FSMWeb which assumes that testers make their tests executable manually, for Mobile Apps many Automatic tools are available to run the test cases, as long as we know which inputs need to be used. For example, tests can be converted to Selenium or any of the other candidate execution environments we identified in Subsection 2.2.2. If we use Selenium, the Appium server executes the Selenium



Table 5.27: Test Path 8 With input values

Edge Id	Constraint	Value	Explanation
A3	O(SelectN), R(SelectM)	selectN = "Trev"	Random selection from name list
	S(SelectN, SelectM)	selectM = "Asprin"	Random selection from medicine list
	Continue-use(SelectN, SelectM)		
I3	O(parName), R(parMed, buttonCE)	parName = "Trev"	Random selection from the database
	S(parName, parMed, buttonCE))	parMed = "Asprin"	Random selection from the database
	continue-use(parName, parMed)	buttonCE = click	Push Cancel Edit Button
I2	O(parName), R(parMed, buttonD)	parName = "Trev"	Random selection from the database
	S(parName, parMed, buttonD)	parMed = "Asprin"	Random selection from the database
	buttonD	buttonD = click	Push Delete Button
A4	R(buttonBack)	buttonBack = click	Push back arrow to exit the app

code and reports results, including pass or fail for each test. Appium is easy to set up and available open-source. Hence, we chose it to execute our test cases for the example. Selenium [23] is an open source software testing framework for web and mobile applications. Selenium provides a test domain-specific language to write the tests such as Java, Python, C# and PHP. Selenium runs on Windows, Linux and MAC. Figure 5.15 shows the Selenium code of Test 8 in Table 5.24. First, the test function calls setUp() in line 4. The setUp() function is shown in Figure 5.14. The

setup() function connects Appium with the mobile device before executing each test case. Desired capabilities specify the set up for the Appium server as well as the test values. This includes the connection type, device name, operating system version (platformVersion), operating system (platformName), app package and app activity. They are sent to the Appium server by the Android Driver via a URL Connection.

Lines 6-9 in Figure 5.15 find the combobox of the patient name then select the name. Selenium will look for the input types in Table 5.2 and save the reference to the input types using the findEelementByAndroidUIautomator function. Then, we can send the input values and the events using the reference of the object such as AddMed.click(). Lines 10-12 find the medicine name combobox and select the medicine to enter for the edit page. Lines 13-15 find the cancel element and perform the cancel action. Line 16 enters the edit page again because we reference the edit page. Lines 17-19 finds the delete button and performs the delete action, then goes back to the main page. Line 20 presses the back button on the Android device. Finally, line 23 disconnects the connection with the mobile app to start a new test case.

The test cases were executed on a Samsung edge 6 phone with Android version 7.0. Figure 5.16 shows the results of the test executions: ten test cases passed and one failed. Test 9 failed because the app was unable to change the patient and medicine name. The execution time for the test suite is 10 minutes. Figures D.1 to D.10 in Appendix D show the selenium code for the remaining test cases. All but one test case passed. Test 9 (in Table D.8) failed. Figure 5.16 shows that test 9 failed.

```

@BeforeClass
public void setUp() throws MalformedURLException {
    DesiredCapabilities capabilities = new DesiredCapabilities();
    capabilities.setCapability(CapabilityType.BROWSER_NAME, "");
    capabilities.setCapability("deviceName", "Ahmed");
    capabilities.setCapability("platformVersion", "7.0");
    capabilities.setCapability("platformName", "Android");
    capabilities.setCapability("appPackage", "family.meds.list");
    capabilities.setCapability("appActivity", "family.meds.list.main");

    wd = new AndroidDriver(new URL("http://127.0.0.1:4723/wd/hub"),
        capabilities);
}

```

Figure 5.14: Setup Connection

```

1  @Test
2  public void Test8() throws MalformedURLException {
3
4
5
6
7      List<WebElement> compoboxName =
8          wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.TextView\")");
9      WebElement selectName = compoboxName.get(0);
10     selectName.sendKeys("Trev");
11     MobileElement EditMed = (MobileElement) wd
12         .findElementByAndroidUIAutomator("new UiSelector().text(\"Asprin : 25 mg\")");
13     EditMed.click();
14     MobileElement Cancel = (MobileElement) wd
15         .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
16     Cancel.click();
17     EditMed.click();
18     MobileElement Delete = (MobileElement) wd
19         .findElementByAndroidUIAutomator("new UiSelector().text(\"Delete\")");
20     Delete.click();
21     wd.pressKeyCode(AndroidKeyCode.BACK);
22     // exit
23     wd.quit();
24 }

```

Figure 5.15: Test 8 in Selenium

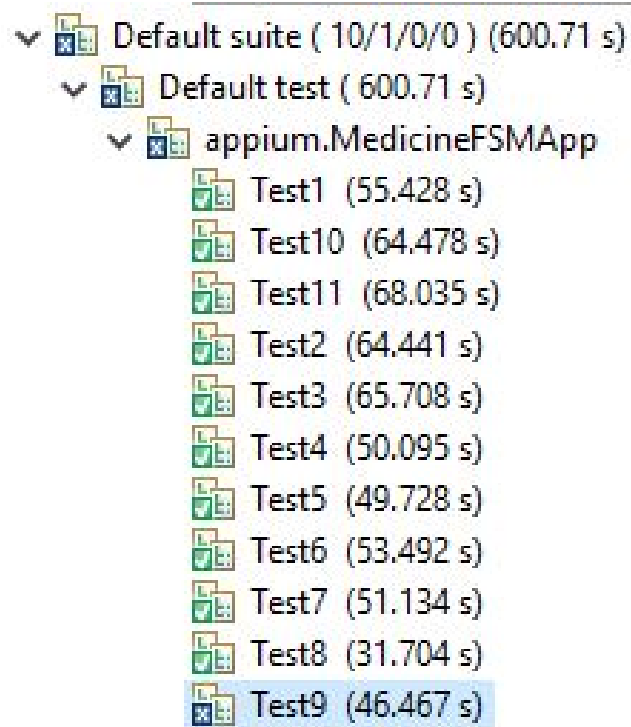


Figure 5.16: Example Execution Results

## Chapter 6

### Comparing FSMApp and other Approaches

In Chapter 2, we identified 2 approaches ([88, 37]) that also perform Black-Box MBT for Mobile Apps. In this chapter, we compare FSMApp with these approaches [88, 37]. Later in Chapter 7, we perform a number of case study comparisons.

The example used here to compare FSMApp and other approaches is the Family Medicine App (Subsection 5.2) which we introduced as our running example to illustrate how the FSMApp method works. Subsection 6.1 introduces the ESG method [88]. Subsection 6.2 describes the GUI Crawling-Based Technique [37]. We apply these techniques to the Family Medicine App. Subsection 6.3 compares the results for FSMApp with these approaches.

#### 6.1 Event Sequence Graph (ESG) Method

de Cleve Farto et al. [88] used an Event Sequence Graph (ESG) to test mobile apps. Their approach consist of the following phases:

1. Create the Event Sequence Graph (ESG) test model. An Event Sequence Graph (ESG) is a directed graph which includes events (nodes)<sup>1</sup> and edges to

---

<sup>1</sup>FSMApp refer to events as actions

connect the events. The nodes "start" and "end" of the graph represent the start and the end of node of the graph. The ESG does not include (multiple) inputs in the graph explicitly rather, they are modeled with decision tables nodes that are associated with nodes that are marked as double circles.

2. Generate paths and implement test cases from the ESG model. de Cleva Farto et al. [88] use TSD4WSC to generate the ESG model and complete Event Sequences (CESs). An CES is a linear sequence test path. The CESs are generated from ESG to cover all edges. Input and output values are determined using the decision table(s) mentioned above. Then, CESs are converted to Robotium. The input selection is ad-hoc from the decision table.
3. Execute implemented test cases with Robotium and collect data. The CESs are executed in the Android Virtual Device (AVD). Execution time is measured and faults are identified.

We apply this method to the Family Medicines List App. Figures 6.1 to 6.3 show the Event Sequence Graph of the Family Medicines List application. We divided the ESG into three figures since even for this small app, the graph is rather large. The double-circled nodes associated with providing a decision table(s). Decision tables (DT) describe type of input data required for test cases and any constraints for value selection[170]. Decision tables help to select inputs for the events of the ESG model. A decision table [59] is defined as  $DT = \{C, E, R\}$  where

- C is a set of constraints with value true, false or do not care.
- E is a set of events.
- R is a set of rules for inputs that causes an event to occur.

This is similar to the input selection constraints in FSMApp. Tables 6.1 to 6.6 shows the DTs of the family medicine list app. For example, Table 6.1 shows the decision table for new medicines. The constraints (C) are Medicine value (parMed), accept button (buttonANMA) and cancel button (buttonANMC). The Events are accept or cancel "add new medicine". The first rule (R1) means if perMed has a value and buttonANMA is clicked, then the medicine name is saved. R2 means regardless of when that perMed has a value and buttonANMC is clicked, then the medicine is cancelled. The decision table also covers error message events (See nodes in Figure 6.1). If a value is selected that does not meet any of the constraint in the decision table or if a required field is not filled in, an error message is sent.

Table 6.1: New Medicines (ESG) Decision Table

		Rules	
		New Medicines	
Constr.	parMed	T	DC
	buttonANMA	T	F
	buttonANMC	F	T
Events	Accept	✓	
	Cancel		✓



Table 6.2: New NameP (ESG) Decision Table

		Rules	
		newNameP	
Constr.	parName	T	DC
	buttonANNPA	T	F
	buttonANNPC	F	T
Events	Accept	✓	
	Cancel		✓

Table 6.3: Dosage (ESG) Decision Table

		Rules	
		Dosage	
Constr.	parDosage	T	DC
	buttonADA	T	F
	buttonADC	F	T
Events	Accept	✓	
	Cancel		✓

Table 6.4: When (ESG) Decision Table

		Rules	
		When	
		R1	R2
Constr.	parWhen	T	DC
	buttonAWA	T	F
	buttonAWC	F	T
Events	Accept	✓	
	Cancel		✓

Table 6.5: Instruction (ESG) Decision Table

		Rules	
		Instructions	
		R1	R2
Constr.	parInstruction	T	DC
	buttonAIA	T	F
	buttonAIC	F	T
Events	Accept	✓	
	Cancel		✓

Table 6.6: Edit (ESG) Decision Table

		Rules	
		Edit	
		R1	R2
Constr.	parName	T	T
	parMed	T	T
	buttonE	T	F
	buttonCE	F	T
Events	Delete	✓	
	Cancel		✓

ESG generates a single test path or multiple test paths that full fills edge coverage. Table 6.7 lists the 17 test paths. It consists of 210 nodes including decision table nodes. The test paths in Table 6.7 provide edge coverage.



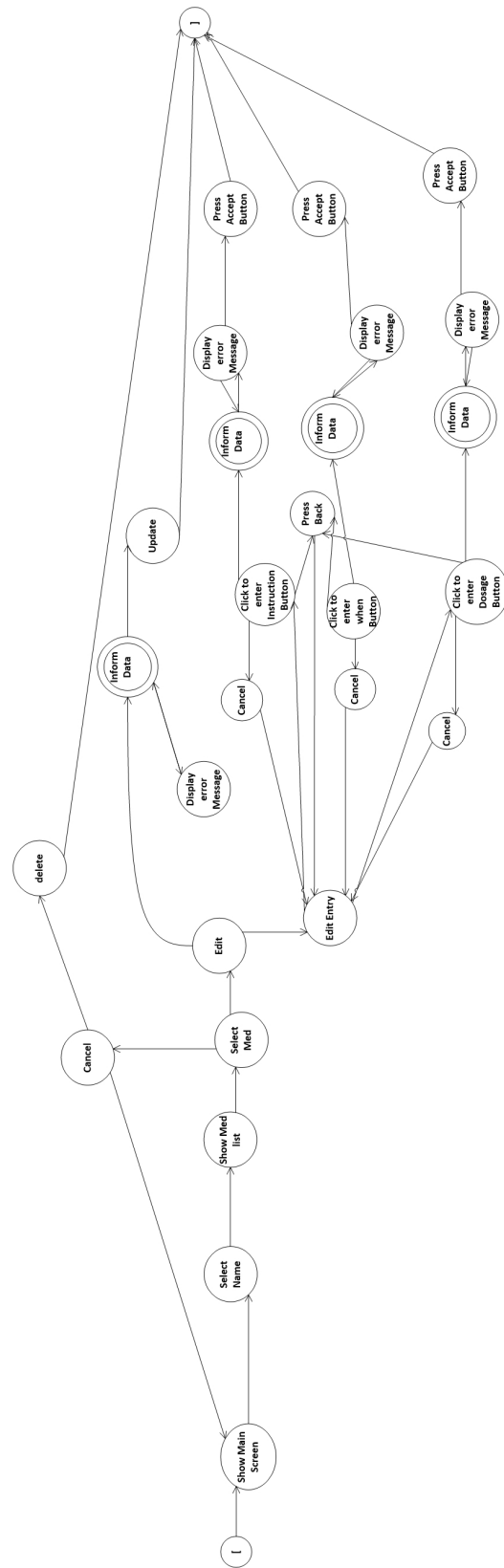


Figure 6.2: Family Medicines List (ESG) Edit Med

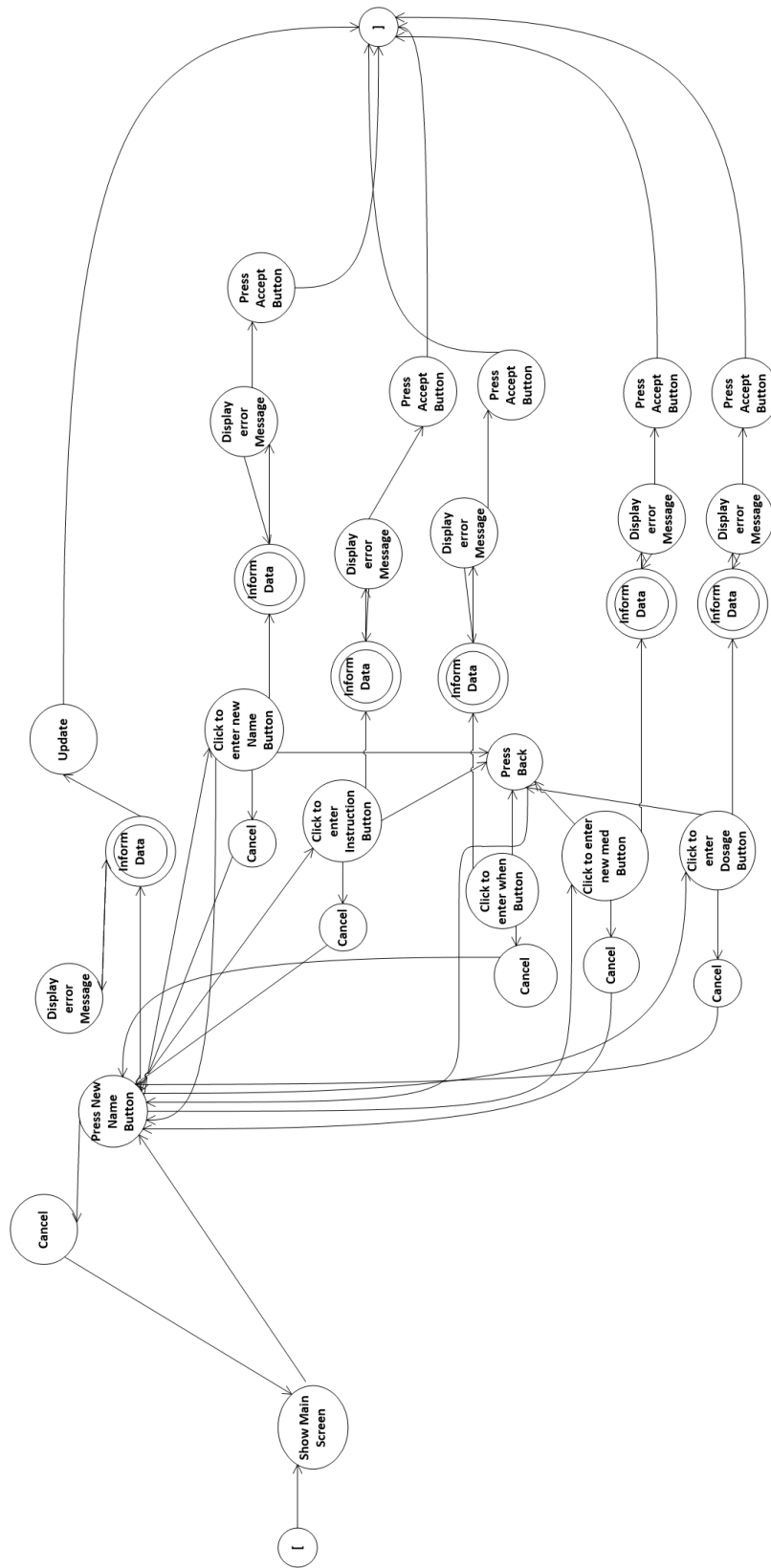


Figure 6.3: Family Medicines List (ESG) Add Patient Name

Table 6.7: Family Medicines List (ESG) Test Paths

<b>Id</b>	<b>Test Path</b>	<b>Length</b>
1	[, Show_Main_Screen, Press_back,]	2
2	[, Show_Main_Screen, Press_Add_New_Med_Button, New_Med, Press_cancel_Button, Show_Main_Screen, Press_Add_New_Med_Button, New_Med, <b>Inform_Data</b> , Display_error_message, <b>Inform_Data</b> , Press_Add_New_Button ,]	11
3	[, Show_Main_Screen, Press_Add_new_med_button, new_med, click_to_enter_instruction_button, cancel, new_med, click_to_enter_instruction_button, press_back, New_Med, Click_to_enter_Instruction_Button, <b>inform_data</b> , display_error_message, <b>inform_data</b> , press_accept_button ,]	14
4	[, Show_Main_Screen, Press_Add_New_Med_Button, New_Med, Click_to_enter_when_button, cancel, New_Med, Click_to_enter_when_button, press_back, new_med, click_to_enter_when_button, <b>inform_data</b> , display_error_message, <b>inform_data</b> , press_Accept_Button,]	14
5	[, Show_Main_Screen, Press_Add_New_Med_Button, New_Med, Click_to_enter_med_button, cancel, new_med, click_to_enter_new_med_button, press_back, new_med, click_to_enter_new_med_button, <b>inform_data</b> , display_error_message, <b>inform_data</b> , press_accept_button,]	14
6	[, Show_Main_Screen, Press_Add_New_Med_Button, New_Med, click_to_enter_dosage_button, cancel, new_med, click_to_enter_dosage_button, press_back, new_med, click_to_enter_dosage_button, <b>inform_data</b> , display_error_message, <b>inform_data</b> , press_Accept_button, ]	15
Continued on next page		

**Table 6.7 – continued from previous page**

<b>Id</b>	<b>Test Path</b>	<b>Length</b>
7	[, Show_main_screen, select_name, show_med_list, select_med, show_med, cancel, show_main_screen, select_name, show_med_list, select_med, show_med ,delete,]	12
8	[, Show_main_screen, select_name, show_med_list, select_med, show_med, edit, edit_entry, press_back, edit_entry, <b>inform_data</b> , display_error_message, <b>inform_data</b> , update, ]	13
9	[, show_main_screen, select_name, show_med_list, select_med, show_med, edit, edit_entry, click_to_enter_dosage, cancel, edit_entry, press_back, edit_entry, click_to_enter_dosage, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept, ]	17
10	[, Show_main_screen, select_name, show_med_list, select_med, show_med, edit, edit_entry, click_to_enter_when, cancel, edit_entry, press_back, edit_entry, click_to_enter_when, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept,]	17
11	[, Show_main_screen, select_name, show_med_list, select_med, show_med, edit, edit_entry, press_back, edit_entry, click_to_enter_instructions, cancel, edit_entry, click_to_enter_instructions, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept, ]	17
12	[, Show_main_screen, press_new_name_button, cancel, show_main_screen, press_new_name_button, <b>inform_data</b> , display_error_message, <b>inform_data</b> , update ,]	9
13	[, show_main_screen, press_new_button, click_to_enter_new_name, cancel, press_new_name_button, click_to_enter_new_name, press_back, press_new_name_button, click_to_enter_new_name, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept, ]	13
Continued on next page		



**Table 6.7 – continued from previous page**

<b>Id</b>	<b>Test Path</b>	<b>Length</b>
14	[, show_main_screen, press_new_name_button, click_to_enter_new_med, cancel, press_new_name_button, click_to_enter_new_med, press_back, press_new_name_button, click_to_enter_new_med, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept, ]	13
15	[, Show_main_screen, press_new_name_button, click_to_enter_new_dosage, cancel, press_new_name_button, click_to_enter_new_dosage, press_back, press_new_name_button, click_to_enter_new_dosage, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept, ]	13
16	[, Show_main_screen, press_new_name_button, click_to_enter_when, cancel, press_new_name_button, click_to_enter_when, press_back, press_new_name_button, click_to_enter_new_when, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept, ]	13
17	[, Show_main_screen, press_new_name_button, click_to_enter_instructions, cancel, press_new_name_button, click_to_enter_instructions, press_back, press_new_name_button, click_to_enter_instructions, <b>inform_data</b> , display_error_message, <b>inform_data</b> , accept, ]	13

de Cleve Farto et al. [88] consider the selection of inputs a constraint satisfaction problem (CSP) [191]. CSP is defined by a set of variables and conditions (C). A variable consider a all possible values of the nonempty domain. The constraint constant of subset of variables and combinations of values which make an event true. Table 6.1 has two CSPs:

- C1 = {parMed = "Aspirin"; buttonANMA= Click}

- $C2 = \{\text{buttonANMC} = \text{Click}\}$

de Cleve Farto et al. [88] execute a test path with CSPs using Robotium framework [22]. We execute the test path on Appium so as to better compare the approaches. The test cases were executed on a Samsung edge 6 phone with Android version 7.0. Table 6.12 shows the summary of ESG execution. The execution time is 15 minutes. All test cases passed except two test paths failed (Test 5 and 8). ESG found one defect which cannot update the patient name and medicine name. Also, ESG tests failed because when the test performs a press back button action to the previous state, the test setup failed, because the app exits instead of going to the previous page.

## 6.2 A GUI Crawling-Based Technique for Android Mobile Application Testing

Amalfitano et al. [37] used a crawler-based automated technique to test mobile applications for crash testing and regression testing. We focus on black-box functional testing without crash testing to compare with FSMApp. Amalfitano et al. [37] described the following phases:

1. Create the model using a crawler-based technique. The technique generates a GUI Tree using an iterative depth-first search. The nodes of the tree represent the user interfaces (mobile screens) of the mobile app and the edges represent the event based transitions between the nodes. The model reaches a leaf node when it encounters an event that causes a loop. The tool captures data of the screens and events. When events are fired, they are written on the edges. Amalfitano et al. [37] also provide a supporting tool  $A^2T^2$  (Android Automatic

Testing Tool). The tool developed in Java with three main components: the Java code instrumentation, the GUI Crawler, and the Test Case Generation. The Java code instrumentation allows java to capture the crashes at runtime. The GUI Crawler generates GUI tress. The Test Case Generation generate the abstract executable testing case with the support of crash testing and regression testing. The tool supports TextView labels, TextEdit fields, Buttons, and Dialogs only.

The GUI crawler relies on two main temporary lists: an event list and an interface list. The event list captures the fired events and the interface list captures the information on the screens. The GUI tree is generated by the following steps:

- (a) When the app is opened, the first screen is considered the root of the tree. The first screen should be described in term of activity instance, widgets, properties and event handlers (See Appendix A). The screen description adds into interface list to distinguish between the mobile screens.
- (b) Find all the interface fireable events, choose random values to fire the event that sets into the widget editable properties and add the description (pre-conditions) into event list.
- (c) Choose one fireable event from the event list with the needed preconditions and fire it.
- (d) Get the current interface and add a node for the interface to the GUI tree and connect the node with an edge between the node and the associated nodes.
- (e) Describe the current interface in terms of activity instance, widgets, properties and event handlers and store the description in the interface list.

Check the current interface is equivalent to previous any visited interfaces or it is new interface. if the current interface is equivalent or it does not have any fireable event, the current interface node will be a leaf. Otherwise, the current interface considers as new interface, describes each events of the interface and add it to the event list. The event to call the current interface removes from the event list.

(f) repeat steps (c)-(e) until the fireable event list is empty

2. Test cases are the event sequences from the root node of the tree to the leaves of the tree. The test cases are generated with the  $A^2T^2$  tool using the Robotium framework<sup>2</sup>. Test cases meet edge coverage.
3.  $A^2T^2$  tool and Robotium framework execute the test cases and captures the injected faults. The test cases are JUnit test cases.

We apply this method to the Family Medicines List App. Figures 6.4 to 6.6 show the tree of the Family Medicines List application. We divided the tree into three figures since even for this small app, the graph is rather large. The actions associated edges are shown in Table 6.8 to keep the tree simple. Table 6.8 has four columns. Columns 1 and 3 represent Edges id on the figures, and columns 2 and 4 represent the action on the edges. The tree has 88 nodes and 88 edges. We generated the model in 44 minutes.

---

<sup>2</sup>The Robotium framework is a testing tool for mobile apps and for analysing the components of Android apps when they are executing.

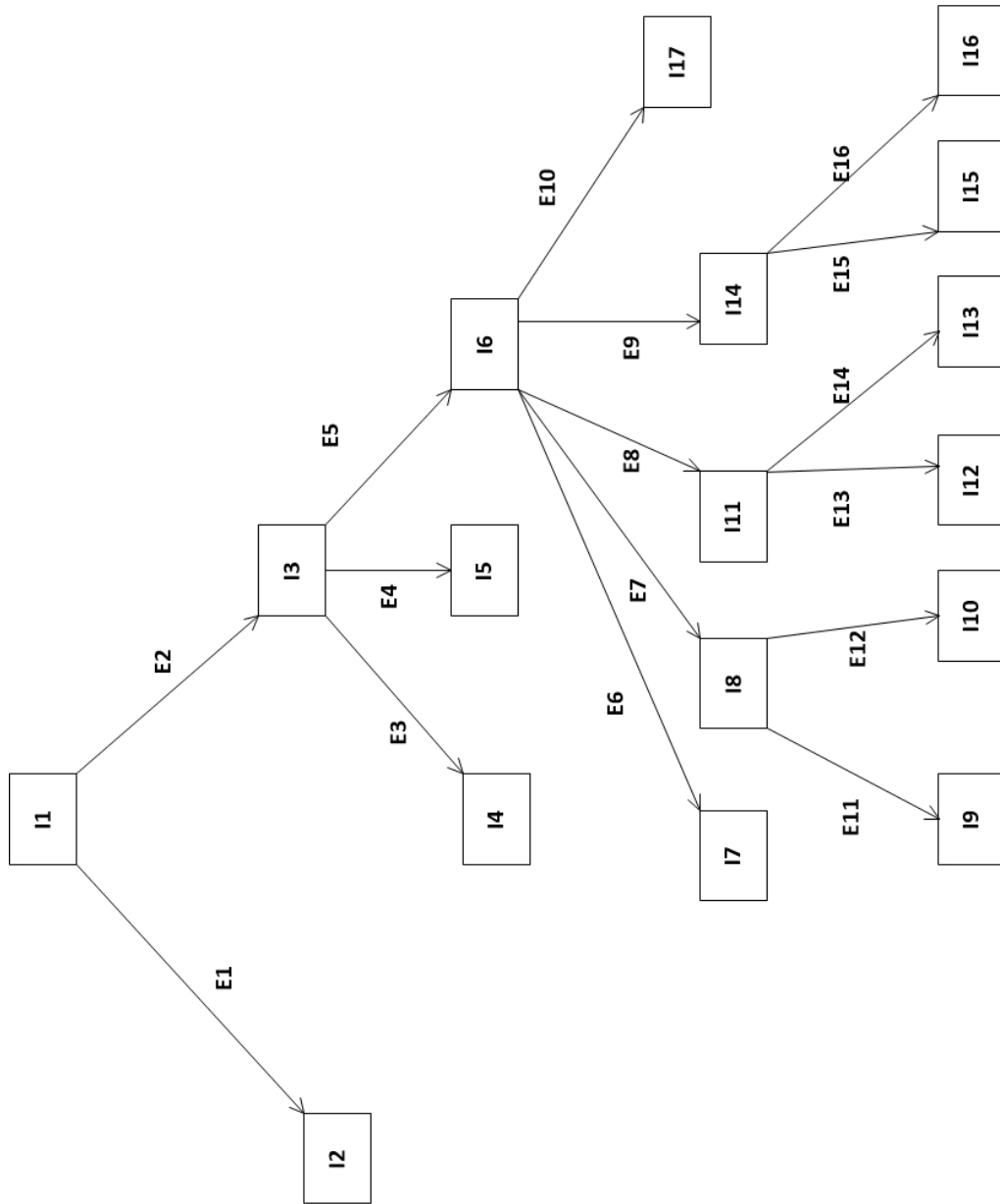


Figure 6.4: A GUI Crawling-Based Technique Model

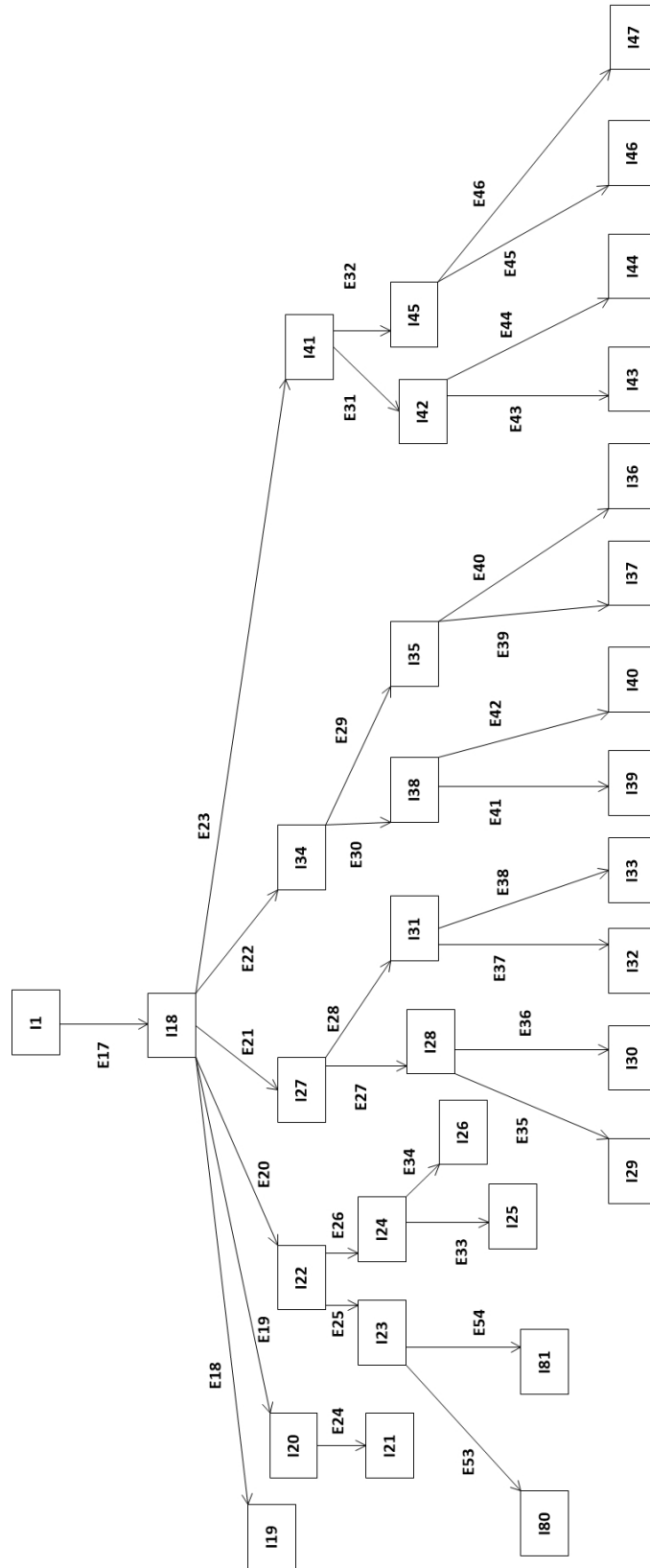


Figure 6.5: A GUI Crawling-Based Technique Model

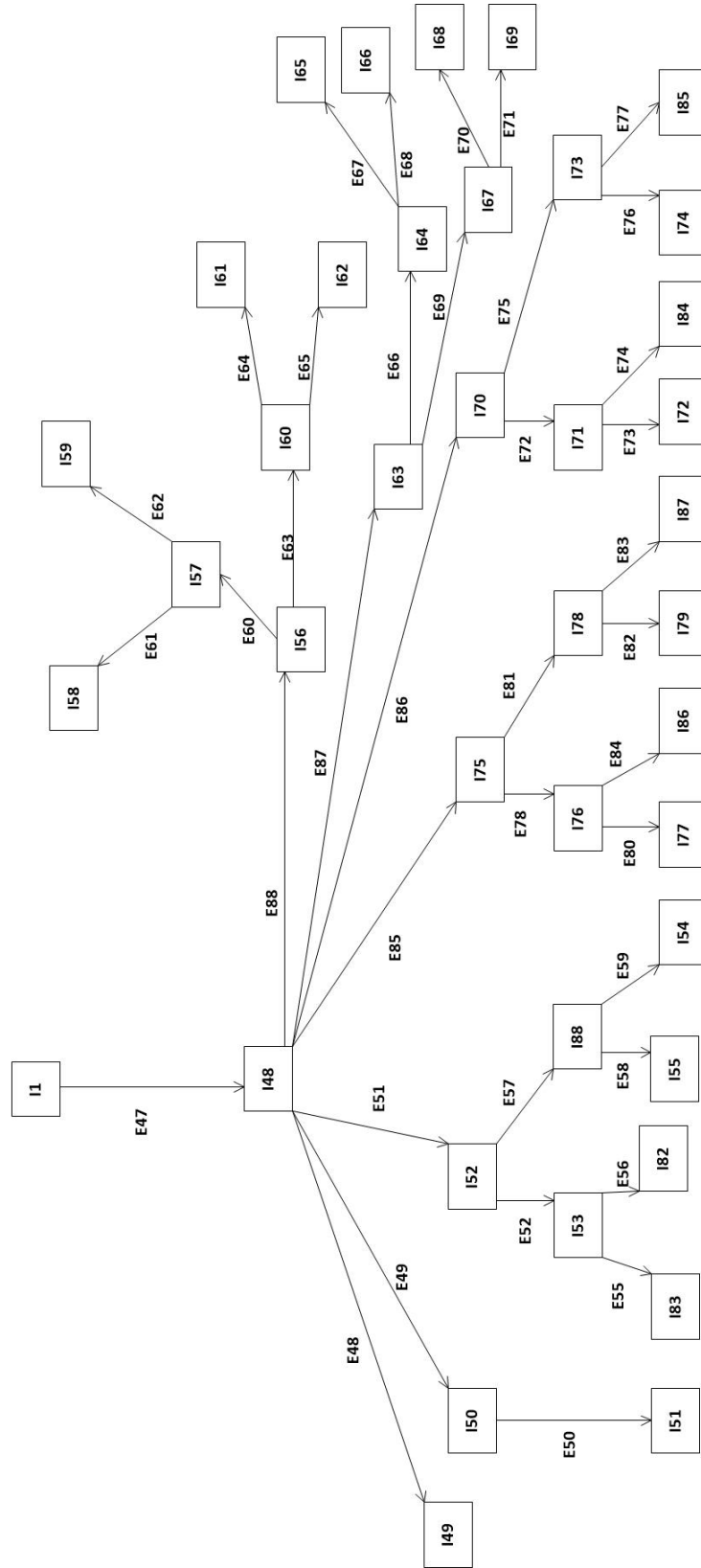


Figure 6.6: A GUI Crawling-Based Technique Model

Table 6.8: A GUI Crawling-Based Technique for Android Mobile Application Testing Edges

Edges	Actions	Edges	Actions
E1	Select Name	E45	Click on Cancel Button
E2	Click on Selected Medicines	E46	Click on Accept Button
E3	Click on Cancel Button	E47	Click on Add New Button
E4	Click on Delete Button	E48	Click on Cancel Button
E5	Click on Edit Button	E49	Click on Add new Button
E6	Click on Cancel Button	E50	Click on OK Button
E7	Click on Dosage Button	E51	Click on New Name Button
E8	Click on When Button	E52	Click on Cancel Button
E9	Click on Instruction Button	E53	Click on Cancel Button
E10	Click on Update Button	E54	Click on Accept Button
E11	Click on Cancel Button	E55	Click on Cancel Button
E12	Click on Accept Button	E56	Click on Accept Button
E13	Click on Cancel Button	E57	Click on Accept Button
E14	Click on Accept Button	E58	Click on Cancel Button
E15	Click on Cancel Button	E59	Click on Accept Button
E16	Click on Accept Button	E60	Click on Cancel Button
E17	Click on Add New Med Button	E61	Click on Cancel Button
E18	Click on Add Cancel Button	E62	Click on Accept Button
E19	Click on Add New Button	E63	Click on Accept Button
E20	Click on Add New Med Button	E64	Click on Cancel Button
E21	Click on Dosage Button	E65	Click on Accept Button
E22	Click on When Button	E66	Click on Cancel Button
E23	Click on Instruction Button	E67	Click on Cancel Button
E24	Click on OK Button	E68	Click on Accept Button
E25	Click on Cancel Button	E69	Click on Accept Button
E26	Click on Accept Button	E70	Click on Cancel Button
E27	Click on Cancel Button	E71	Click on Accept Button
E28	Click on Accept Button	E72	Click on Cancel Button
E29	Click on Cancel Button	E73	Click on Cancel Button
E30	Click on Accept Button	E74	Click on Accept Button
E31	Click on Cancel Button	E75	Click on Accept Button
E32	Click on Accept Button	E76	Click on Accept Button
E33	Click on Add Cancel Button	E77	Click on Cancel Button
E34	Click on Accept Button	E78	Click on Cancel Button
E35	Click on Add Cancel Button	E79	Click on Cancel Button
E36	Click on OK Button	E80	Click on Accept Button
E37	Click on Add Cancel Button	E81	Click on Accept Button
E38	Click on Accept Button	E82	Click on Accept Button
E39	Click on Add Cancel Button	E83	Click on Cancel Button
E40	Click on Accept Button	E84	Click on Cancel Button
E41	Click on Add Cancel Button	E85	Click on New Med Button
E42	Click on Accept Button	E86	Click on Instruction Button
E43	Click on Add Cancel Button	E87	Click on When Button
E44	Click on Accept Button	E88	Click on Dosage Button



Table 6.9 lists test cases. Columns 1 and 2 show the test id. Columns 2 and 4 show the test paths. Columns 3 and 6 show the length of test paths. The Crawler-based approach generated 51 test paths with a total length of 240 steps. Table 6.10 shows the unique inputs for test cases. The total inputs and actions for test cases are 272. Most of the inputs are repeated because we have many test paths.

Table 6.9: Family Medicines List (GUI Crawling-Based Technique) Test Paths

Id	Test Path	Length	Id	Test Path	Length
1	[I1, I2]	2	26	[I1, I18, I41, I42, I43]	5
2	[I1, I3, I4]	3	27	[I1, I18, I41, I42, I44]	5
3	[I1, I3, I5]	3	28	[I1, I18, I41, I45, I46]	5
4	[I1, I3, I6, I7]	4	29	[I1, I18, I41, I45, I47]	5
5	[I1, I3, I6, I8, I9]	5	30	[I1, I48, I49]	3
6	[I1, I3, I6, I8, I10]	5	31	[I1, I48, I50, I51]	4
7	[I1, I3, I6, I11, I12]	5	32	[I1, I48, I52, I53, I82]	5
8	[I1, I3, I6, I11, I13]	5	33	[I1, I48, I52, I53, I83]	5
9	[I1, I3, I6, I14, I15]	5	34	[I1, I48, I52, I88, I54]	5
10	[I1, I3, I6, I14, I16]	5	35	[I1, I48, I52, I88, I55]	5
11	[I1, I3, I6, I17]	4	36	[I1, I48, I56, I57, I58]	5
12	[I1, I18, I19]	3	37	[I1, I48, I56, I57, I59]	5
13	[I1, I18, I20, I21]	4	38	[I1, I48, I56, I60, I61]	5
14	[I1, I18, I22, I23, I80]	5	39	[I1, I48, I56, I60, I62]	5
15	[I1, I18, I22, I23, I81]	5	40	[I1, I48, I63, I64, I65]	5
16	[I1, I18, I22, I24, I25]	5	41	[I1, I48, I63, I64, I66]	5
17	[I1, I18, I22, I24, I26]	5	42	[I1, I48, I63, I67, I68]	5
18	[I1, I18, I27, I28, I29]	5	43	[I1, I48, I63, I67, I69]	5
19	[I1, I18, I27, I28, I30]	5	44	[I1, I48, I70, I71, I72]	5
20	[I1, I18, I27, I31, I32]	5	45	[I1, I47, I70, I71, I84]	5
21	[I1, I18, I27, I31, I33]	5	46	[I1, I48, I70, I73, I74]	5
22	[I1, I18, I34, I35, I36]	5	47	[I1, I48, I70, I73, I85]	5
23	[I1, I18, I34, I35, I37]	5	48	[I1, I48, I75, I76, I77]	5
24	[I1, I18, I34, I38, I39]	5	49	[I1, I48, I75, I76, I86]	5
25	[I1, I18, I34, I38, I40]	5	50	[I1, I48, I75, I78, I79]	5
			51	[I1, I48, I75, I78, I87]	5

Table 6.10: Parameter for GUI-Crawling-based Technique

Name	Value
Select Name	Trev
Select Med	Aspirin : 25 mg
Dosage	25 mg
When	Everyday
New Med	FIBER
New Name	Ahmed
Instruction	One Per Day

Table 6.12 shows the summary results of the Crawler-based approach execution. The execution time is 23 minutes. Crawler-based results are 50 passed and one failed (Test 31). The approach found one defect which cannot update the patient name and medicine name.

### 6.3 Comparison of Results

We will compare FSMApp, ESG and the Crawler-based techniques with respect to models, path generation, input generation and test execution.

Table 6.11 compares FSMApp, ESG and the Crawler-based techniques with respect to model building, test generation, input selection, and making tests executable and test execution. During model building, input constraints are represented on the edges in FSMApp as compared to an event list in the Crawler-based technique whereas ESG represents them with double circles node with an associated decision table. Actions are represented on the edges in both FSMApp and the Crawler-based technique, but in ESG they are represented as a node. App Screen (Active) is a LAP in FSMApp, but a node in ESG and the Crawler-based technique. FSMApp, ESG and the Crawler-based approaches navigate the model of the mobile application by edge traversal.

Clusters are only available in FSMApp. FSMApp generates the test paths by first generating paths for each cluster FSM, and then aggregating paths based on coverage criteria. ESG and Crawler-based approaches use a tool to determine the test paths. Only FSMApp removes dummy nodes from test paths to reduce test sequences. Input generation for test sequences in FSMApp is based on resolving input-action constraints. ESG uses ad-hoc selection from the decision table. In case of an incorrect value or a lack of filling a field an error message is sent. The Crawler-based approach generates the input randomly and stores them in the event list.

FSMApp does not require to use any particular tools to make tests executable and execute them. Subsection 5.5 showed how to automate this step using Selenium/Appium. ESG and the Crawler-based approaches run their tests using Robotium.

Table 6.12 compares the results of FSMApp, ESG, and Crawling approaches on the Family Medicines App. Column one and two identify the four phases of test generation and execution. Phase 1 is model building. Phase 2 is test sequence generation, Phase 3 is input selection, and Phase 4 is execution and validation of tests. Column three shows units of comparison. For model generation, we compare model size in terms of nodes, edges, and clusters, as well as model generation time. For test sequence generation, we compare the size of the test sequence in terms of the number of sequences and the total number of test steps. We also compare the time it takes to generate them (in minutes). For input selection, we compare the number of inputs and actions, as well as input generation time. Finally, for test execution, we compare how much test code needed to be written, the number of tests that failed/ passed and the number of defects found. We also compare execution time. Column 4-6 show the results for FSMApp, and ESG [88], and the Crawler-based

technique [37], respectively.

We compare the results in Table 6.12 for each phase:

- Model generation: The FSMApp model is far smaller than both ESG and the Crawler-based technique in terms of nodes and edges. Building the model for FSMApp takes much less time (12 versus 29 and 44 minutes), respectively. One reason for this is of course that the model for FSMApp is much smaller, but the cluster also required fewer repeated nodes and edges. The Crawling-based technique and ESG have the same number of nodes and edges, whereas FSMApp has half the number of nodes because the clusters reduce the number of the nodes and the edges instead of repeating them.
- Generation of test sequences: FSMApp generated 11 test sequences compared to 17 and 51 test sequences, respectively. The Crawler-based approach generates a large number of test sequences because the approach does not support loops, and stops when it encounters a repeated screen. The FSMApp has significantly fewer total test steps (87 steps versus 219 and 240, respectively). The ESG and the Crawler-based approach have far more steps because these approaches result in many repeated actions and have many more test sequences. The Crawler-based approach takes the least time to generate test sequences, because it is easy to follow the path from the root to the leaf of the tree. FSMApp takes 31 minutes (compared to 88 and 15 for the other two) because we generate the test sequences for each cluster, aggregate them, and then perform test step reduction.
- Input selection: The total number of inputs and actions is comparable for ESG and FSMApp, with 135 versus 150 inputs and actions. The Crawler-based approach requires 272 inputs and actions. They take 30 minutes to

generate. The time is large compared to other approaches because there are so many repeated inputs and many more test sequences.

- Execution time: The test code required for FSMApp and ESG are roughly comparable (547 versus 635 LOC), whereas the Crawler-based approach needs almost three times as many test LOC as FSMApp. Due to the large number of test and inputs, this approach also needs more than twice the execution time of FSMApp. ESG has a slightly higher execution time (15 compared to 11 minutes). They each find one defect. However, an additional of the ESG tests failed because when the test performs a press back button action to the previous state, the test setup failed, because the app exits instead of going to the previous page.

Next, we compare the overall time for testing (last row). ESG and the Crawler-based approach take the same time with 113 minutes. This is much longer than for FSMApp which only takes 76 minutes. This is because model generation, time to choose inputs, and execution time for test cases required much less time than the other two techniques. We can therefore conclude that FSMApp is more efficient than the other approaches.

Table 6.11: Comparison of FSMApp and ESG

Model building	FSMApp	ESG	Crawler-based
Input Variables ( $\vec{I}$ )	part of edge annotation $P(\vec{I}, a)$	not explicit in node, annotated node with double circle. separate decision table	Random and saved in the event list
Action (Event) (a)	$P(\vec{I}, a)$	node (annotated w/specific action value) implicit in node	$P(\vec{I}, a)$ implicit in node
App screen & node type	LAP		
Navigation	edge	edge	edge
Cluster	node type: cluster	N/A	N/A
Test generation			
Cluster	local paths	N/A	N/A
Coverage Criteria	node, edge, n-edge, prime path	N/A	N/A
Test Paths	multiple paths via path aggregation table	from start node to end node and reduced using a solution to the Chinese Postman Problem	from start node to end node
Coverage criteria	all combinations, each choice, and base choice		edge coverage
Input Selection	resolve P along test path	ad-hoc from decision table, manual	random and stored in event list
Executable tests	manually; can use selenium/Appium	Robotium	Robotium
Test execution	manually; by tester exercising app can use selenium/Appium	By tester running Robotium	By tester running Robotium

Table 6.12: Comparison of Applying Techniques in Family Medicines List App

Step Id	Phase	Units of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	45	88	88
		Number of edges	87	145	88
		Number of clusters	9	0	0
		Generation Time	14	29	44
2	Generate Test Sequence	Number of test Sequences	11	17	51
		Total test steps	87	219	240
		Generation time	31	38	16
	Total time (1) + (2)		45	67	60
3	Input Selection	#inputs	48	40	83
		#actions	87	110	189
		Time to choose input	20	22	30
	Total time (1) + (2) + (3)		65	98	90
4	Execute Test Cases	Test lines of code	547	635	1503
		Execution time	11	15	23
		Number of failed tests	1	2	1
		Number of passed tests	10	15	50
		Number of defects	1	1	1
	Total time		76	113	113

## Chapter 7

### FSMApp Validation via Case Studies

#### 7.1 Rationale

As we explained in the approach, we provided an example mobile app to illustrate how the FSMApp approach works. Now, we would like to compare and validate the FSMApp with the other approaches for testing mobile applications. The case studies cover a number of mobile apps from different domains and with different sizes.

#### 7.2 Case Study Objectives

We propose to investigate the applicability, scalability, efficiency and effectiveness of FSMApp for testing mobile applications. Furthermore, we want to know how FSMApp compares [88, 37] in these evaluation areas.

#### 7.3 Preparation for Case Study

Before we executed the case studies, we studied each approach and applied all examples in the research papers using each method. Then, we applied all approaches



to the small example in chapter 6. This was done to reduce learning effects that might bias the time needed to test each case study mobile app. We also carefully studied all functions for each of the 10 mobile apps, before applying the approaches. This was done so the time for learning how all functions of each mobile app work would not confound testing time for the mobile app. Learning effects for applying testing methods are more likely for the example rather than the case studies.

## 7.4 Case Study Research Questions

The research questions derived from the case study objectives fall into two categories: those related to FSMAApp and those related to comparison with the two other techniques. Research Question RQ1-RQ4 deal with FSMAApp, while RQ5 emphasize comparison studies.

- RQ1: Applicability. Can we apply FSMAApp to a variety of mobile apps in different application domains and of different sizes? Android Play [11] presents the top categories of mobile apps in the store: Photography, Family, Music & Audio, Entertainment, Shopping, Personalization, Social and Communication. AppBrain [6] presents slightly different top apps categories in the play store: Education, Business, Lifestyle, Entertainment, Music & Audio, Tools, Books & Reference, Personalization, Health & Fitness and Productivity. Some of the top categories overlap. The apps used in our case studies are taken from different categories to show the applicability of the FSMAApp.
- RQ2: Scalability. How does FSMAApp scale when models become larger? The case study includes multiple apps in the same category because we would like to test the scalability of the FSMAApp with different sizes. The apps size on

the disk ranges from 1.18MB to 43.3MB. However, the size of the app does not necessarily correlate with the model size needed for MBT. The case study compares the model size with respect to number of the nodes, and number of edges.

- RQ3: Efficiency. How efficient is FSMAApp? This is evaluated by steps in the test generation process, thus relates to efficiency of models (size), length of test paths and test suite, and test execution effort.
- RQ4: Effectiveness. How effective is FSMAApp at finding defects for Mobile Apps of different sizes and in different domains? The case study executes the test cases and captures the number of the defects.
- RQ5: The following sub-questions compare FSMAApp with other approaches using the same measurement for RQ1-RQ4.

RQ5.1: How does FSMAApp compare to the other methods in terms of applicability?

RQ5.2: How does FSMAApp compare to the other methods in terms of scalability?

RQ5.3: How does FSMAApp compare to the other methods in terms of efficiency?

RQ5.4: How does FSMAApp compare to the other methods in terms of effectiveness?

## 7.5 Units of Analysis

Table 7.1 shows the measurement units on every phase. Column one presents the phase of FSMAApp, and column two presents the measurement. These are the same metric, we used in Subsection 6.3.

Table 7.1: Units of Analyses

Phase	Measurement
Generate Model	Size (#Nodes, #Edge, #Clusters), time to generate the model
Generate Test Sequences	Size (Number of test sequences, total test steps), time to generate the test sequences
Input selection	#inputs, #actions, Time to choose input
Execute Test Cases	Test line of code, Execution time, #fail, #success, #defects

## 7.6 Case Study General Descriptions & Rationale

We have identified 10 Android Mobile Applications as candidates for our case study.:

- Family Medicines List [9] is a student project for medical information. It manages medications by category, dosage, and special instructions. It also prioritizes medications.
- Memory Game Application [13] is a game where pairs of matched images need to be found. The player wins if he matches all images.
- Timber Music Player [28] is an open source Android application for playing music. It does not download music. The features of the app are browse songs, albums and artists, create and edit playlists. It supports six different playing

styles, and provides homescreen widgets. The user can browse device folders, and customize theme and the user interface. Timber supports lyrics for the music and shows the playing queue in the notification of your smartphone.

- An Explorer File Manager (File Explorer) [10] is an all-in-one file management tool. It is an open source Android application. It is a simple, small, fast and efficient file explorer and one of the best file manager apps of the Google Play Store. It is the only file manager to support RTL and to show the size of all folders stored. File Manager is designed for all Android devices including Phones, Tablets, and Android TV. It supports all Android versions from Jellybean, KitKat, Marshmallow to Nougat. The Top Features are a full featured file manager, a smart library file explorer, an external storage file manager, a secure super file manager, a root file manager, an app manager, a process manager, and a network file manager. One can transfer files via FTP from a phone to a PC. The document editor allows editing files on the go. It supports all types of text files such as HTML, XHTML, TXT etc.
- ML Manager [16] is an open-source customizable Android application package (APK) manager for Android. It allows to extract any installed mobile applications, to mark them as the favorite, and to share mobile application (.apk) files. Further features include Extract any installed and system apps and save them as APK, including a batch mode to extract multiple APKs at the same time, the ability to share any APK using other apps like Telegram, Dropbox, email, etc. It organizes your apps by marking them as favorite making access easier, it uploads your latest APKs to APKMirror, it Uninstalls any installed app. Customization is available in settings, including a dark mode, customizing main colors and more, and no root access required.

- Simple Calendar [24] is an open source Android application with optional protocol calendar synchronization (CalDav). The Simple Calendar allows the user to create recurring events, reminders and displays a week id. It has different views of events in the calendar: a monthly view and an event list widget where you can customize the color of the text and the color of the background.
- Amaze File Manager [1] is an open source Android application. It manages the files on a mobile device with basic features like cut, copy, delete, compress, extract etc. Additional features provide multiple tabs, multiple themes, a navigation drawer for quick navigation, an App Manager to open, backup, or directly uninstall any app. One can quickly access history. It provides bookmarks. One can also search for a file. For advanced users, Root explorer provides AES Encryption and Decryption of files for security, and Cloud services support. It also supports building a database and provides a database reader, a Zip/Rar Reader, Apk Reader, and Text Reader.
- Todo List [30] is a simple todo list manager that can only add and delete tasks.
- Minimal ToDo [14] is an open source Android App, a very light and useful app, allowing you to manage a ToDo List easily and quickly. Both Minimal ToDo and Todo List [30] have the same functions: add and delete. Minimal ToDo has a different style and forms to manage the task list. Also, Minimal ToDo has a notification reminder and mark the task as done.
- MIRAKEL: Task Management [15] is an open source Android application. It manages tasks with the following features: create tasks super fast with keywords, smart prioritization for the current day or week, as well as for

overdue tasks.

Table 7.2 summarizes the ten Android mobile applications for the case study. The first column shows the name of the application. The second and the third columns show the number of user reviews and the rating of the application in Google Play. The fourth and fifth columns are related to the Android version that supports the App. The sixth and seventh columns show the size of the application code in terms of download size and size on disk. The eight column gives the type of the application. The last two columns provide the last update to the mobile application, and how many times the application was installed on mobile devices.

The case studies cover seven domains: calendar, simple game, todo list, task management, music manager and File manager. We selected case studies that are Android open source apps with a review rating of 4 or over. The last three rows in Table 7.2 describe case studies that are available with source code from the Titanium development tool [29] or from student projects. The apps differ in size from small to larger to so we can validate the scalability of the FSMAApp.

Table 7.2: Android Mobile Applications

Apps	User Re- view	Rate /5	Android	Current version	Download Size	Code Size (on Disk )	type	last update	installs	category
Family Medicines List [9]	Sample Code							02/25/2018		Health & Fit- ness
Memory Game Application [13]	Sample Code							02/25/2018		Game
Timber[28]	3044	4.3	4.1 and up	1.6	7.87 MB	5.14 MB	Music Player	12/12/2017	100k - 500k	Music & Audio
File Manager [10]	1303	4.1	4.2 and up	3.7	2.32 MB	1.18 MB	File Manager	11/19/2017	100k - 500k	Tools
ML Man- ager [16]	1345	4.6	4.1 and up	3.3	2.47 MB	3.48 MB	APK Ex- tractor	01/23/2018	50k - 100k	Tools
Simple Calen- dar [24]	2634	4.3	4.1 and up	3.3.2	3.13 MB	4.32 MB	Calendar	02/22/2018	500k - 1M	Tools
Amaze File Manager [1]	9796	4.3	4.0 and up	3.2.1	4.59 MB	5.8 MB	File Manager	08/22/2017	500k - 1M	Tools
Todo List [30]	Sample Code							02/25/2018		Productivity
Minimal To Do[14]	725	4.5	4.1 and up	1.2	2.14 MB	12.3 MB	ToDo List	09/23/2015	10k - 50k	Productivity
MIRAKEL: Task Manage- ment [15]	281	4	4.0 and up	3	4.78 MB	43.3 MB	Task Manage- ment	07/29/2015	10k - 50k	Productivity

## 7.7 Case Study Results & Discussion

### 7.7.1 RQ1: Applicability

We applied FSMApp to ten mobile applications from different categories. The apps fall into five categories: Health & fitness, Game, Music & Audio, Tools, and

Productivity. We were successfully able to apply FSMApp to all ten mobile applications. Tables 7.7 to 7.15 report our results. They are organized the same as Table 6.12. They show the results for each phase: model generation, test sequence generation, input selection, and test case execution. Column 3 of Tables 7.7 to 7.15 shows the results of FSMApp. We successfully applied for FSMApp all these different categories app.

### 7.7.2 RQ2: Scalability

We apply FSMApp to mobile apps ranging in size and application domain as shown in Tables 7.7 to 7.15. We compare the FSMApp scalability with regards to four phases: Generate Model, Generate test sequence, input selection, and test execution and validation. Figure 7.1 shows the total number of edges versus the time to generate the model. Figure 7.1 shows the time increases slowly but far the app whose model has 200 edges and the time reaches 60 minutes then go down to 30 min for a model with more edges. The time reaches 60 minutes because the behavior of the app is different and learning the app function for the first time needs more time. The number of components affects on the number of edges as shown in Table 7.3. MIRAKEAL app has 78 buttons. It means that we need more time to draw the connection between them. In general, since the tester's performance in the model build is measured, learning effects can occur.

Figure 7.2 shows the total number of test sequences versus the time to generate the test sequences. Figure 7.2 shows the time first decreases then it increases highly. There are two spikes. These apps have more clusters, and components. The second phase can be effect by loops and the number of edges.

Figure 7.3 shows the total number of test inputs and actions versus the input



selection time. The time increase linearly for less than 180 inputs and actions. It is increasing faster after 320 inputs. This depends on the number of components and types of inputs. The spike in the middle is related to clusters that have a lot of inputs which we do not consider them as nodes.

Figure 7.4 shows the test LOC versus the execution/validation time. The execution/validation time increases rapidly with more than 1000 LOC. For Lower LOC, the tests need less than 10 minutes to execute the test sequences. The time depends on the number of actions and how long it takes the mobile device to response. It is important to know that even for the larger apps the execution of the tests is well below two hours.

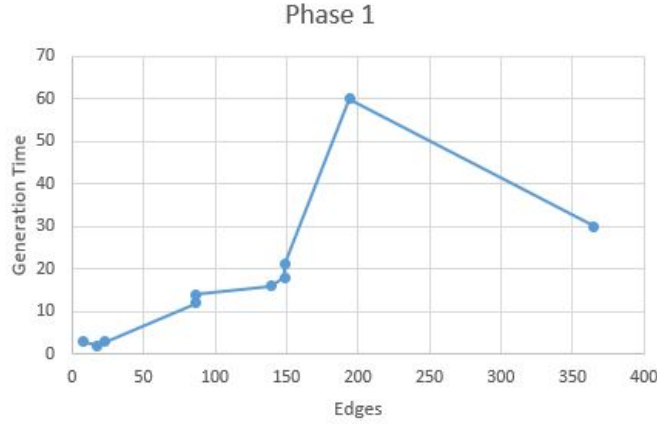


Figure 7.1: Generation Time vs Number of Edges

### 7.7.3 RQ3: Efficiency

We studied the efficiency of FSMAApp. The efficiency is evaluated in the test generation process, length of test sequences, and test execution effort (time). Table 7.4 presents a summary for each app model size and time for generation model phase. Column 1 shows the name of the apps. Columns 2-4 show the number of nodes, edges, and clusters of the model, respectively. The last column shows the

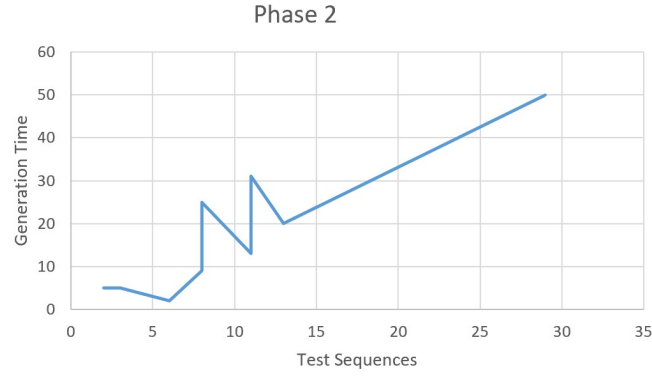


Figure 7.2: Generation Time vs Number of Test Sequences

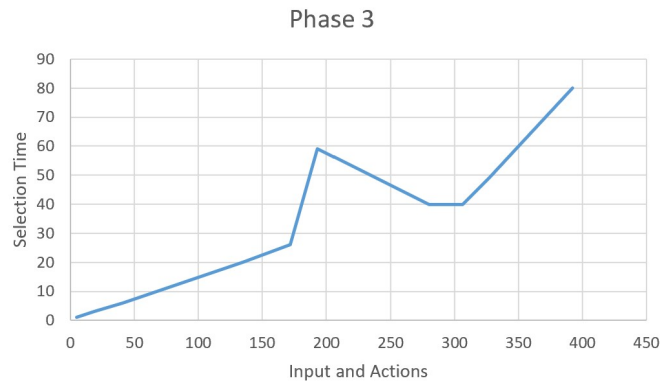


Figure 7.3: Input Selection Time vs Inputs and Actions

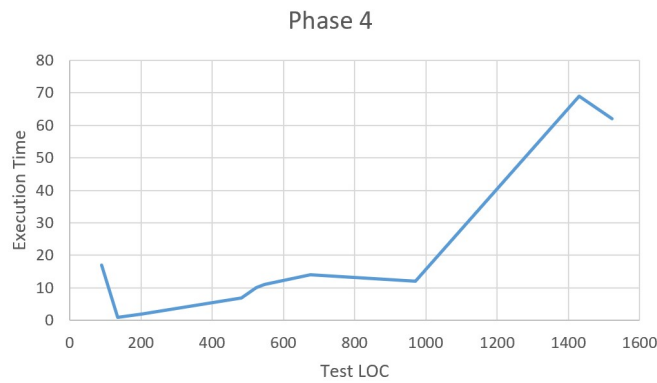


Figure 7.4: Execution/Validation Time vs Test LOC

model generation time. The model generation time is measure by the designer while drawing the model. Game Memory App [13] has the smallest model with six nodes

and eight edges and no cluster. Therefore, MIRAKAL app [15] has the largest model with 167 nodes, 465 nodes, and 12 clusters.

The model generation time of the ten apps took between 2 to 60 minutes. Tool Category has four apps: File Manager [10], ML Manager [16], Simple Calendar [24], and Amaze [1]. The four apps have vary in the model size in term of nodes, edges and clusters. Simple Calendar has the smallest model size with 82 nodes, 149 edges, and 5 clusters, whereas File Manager, has the largest model with 188 nodes, 194 edges, and 18 clusters. The difference between the two models in term of size is around 30% for nodes and edges. File Manager app has the maximum model generation time because it has 18 clusters. The clusters require more time to identify them and find the connection between them. We also include three apps from the productivity category: Todo list [30], Minimal Todo [14] and task Management (MIRAKEL) [15]. Todo List has the smallest model with 12 nodes, 18 edges and one cluster whereas MIRAKEL app has the largest model with 167 nodes, 365 edges and 12 clusters. The difference between the two models in term of size is 90%. FSMApp can be applied to the same category app with many different sizes.

Table 7.4: Model Size Summary

App	Nodes	Edges	Clusters	Model Time
Family Medicines	45	87	9	14
Memory Game	6	8	0	3
Timber	52	87	8	12
File Manager	118	194	18	60
ML Manager	87	149	7	18
Simple Calendar	82	149	5	21
Amaze	83	139	11	16
Todo list	12	18	1	2
Minimal	15	23	2	3
MIRAKEL	168	365	12	30

Table 7.5 shows the results of applying FSMApp on ten mobile apps. Column one indicates mobile apps name. Columns 2-4 show the number of test sequences, number of steps (after reduction) and the time (in minutes) for generating test sequence. The time is calculated by the tester. The includes generation of cluster test paths and the aggregation test sequences. Columns 5-7 show the number of inputs, number of actions, and time to choose inputs (in minutes) for the input selection phase. The time is calculated by the tester. The time includes identification of inputs boundaries and input selection to execute the test sequences. Column 8 shows the execution time in minutes (the time is measure with the tool) and Column 9 shows the total time in minutes for all phases: model generation, test cases generation, input selection, and test case execution. The last column shows the time per step in seconds  $((\text{total time} / \text{number of steps}) * 60)$ . We calculate the

step time to know the average time of testing the apps because we have different sizes with the model and the size of app in the disk. By the number of steps, we can estimate the average of total time for the future case studies.

The time range of step for all apps is between 27-56 seconds except the game app required 172 seconds per step. The game app took a long time per step because the implementation of the game has the images with a sources id. In this case, we use a loop to check all the photos. The average time to execute each step is 42 seconds. The smallest model required 48 seconds for each step. The largest model needed 41 seconds to execute the step, and the second largest model required 35 seconds. We can conclude that FSMAApp is efficient for the large model as small models.

Table 7.5: Summary of Test Generation and Execution

Apps	#Tests	#Steps	Time	#Inputs	#Actions	Choose Time	Execution Time	Total Time	Step Time
Family Medicines	11	87	31	48	87	20	11	76	52
Game Memory	6	8	2	0	5	1	17	23	172
Timber	8	120	9	3	190	59	7	87	44
File Man- ager	29	381	50	11	381	80	69	259	41
ML Man- ager	8	122	11	2	170	26	10	65	32
Simple Calendar	8	105	25	30	276	40	12	98	56
Amaze	11	188	13	14	266	40	14	83	27
Todo list	2	17	5	3	16	3	1	11	39
Minimal	3	20	5	6	35	6	2	16	48
MIRAKEL	13	277	20	19	308	49	62	161	35

#### 7.7.4 RQ4: Effectiveness

The case study executes the test cases and captures the number of defects. Table 7.6 shows the execution results for ten apps. Column one shows the name of the mobile application. Columns 2 and 3 show the number tests that failed and passed, respectively. The last column shows the number of defects. All the apps do not show any defect except Family Medicines list app has one defect. This may be because we selected highly rated apps (see Subsection 7.5) and hence they are not likely to show many defects. The only except is the Family Medicines List app which was developed by student. Hence, with our selection of case studies, we were not able to show full effectiveness.

Table 7.6: Defect Summary

App	#Failed	#Passed	#Defect
Family Medicines	1	11	1
Game Memory	0	1	0
Timber	0	8	0
File Manager	0	14	0
ML Manager	0	8	0
Simple Calendar	0	8	0
Amaze	0	11	0
Todo list	0	1	0
Minimal	0	3	0
MIRAKEL	0	13	0

## 7.7.5 Compare FSMApp with Other Approaches

### 7.7.5.1 RQ5.1: Applicability

FSMApp can be applied in many domains and for varying sizes of mobile apps as show in Subsection 7.7.1. We applied ESG approach of the same mobile app with different categories. Tables 7.7 to 7.15 show the result of ESG approach on Column 5 and the Crawler-based approach on Column 6. We applied the ESG and the Crawler-based approach to all ten mobile apps using four phases: model generation, test sequence generation, input selection and test case execution. The Crawler-based approach cannot be used in different categories. We cannot apply the Crawler-based approach to the game category because the approach does not support loops. Also, the Crawler-based approach reach the leaf of the tree when visit the same mobile screen. When we tried to create the model, we stopped when we flip the screen because we reach to the same screen. FSMApp and ESG can be applied to any mobile category, whereas the Crawler-based approach cannot apply to all categories.

### 7.7.5.2 RQ5.2: Scalability

We compare the scalability of FSMApp, ESG, and the Crawler-based approach with the model size in terms of edges, model generation time, number of test sequences, generation test sequences time, total of inputs and actions, time to choose input, test lines of code and execution time. Tables 7.7 to 7.15 show the comparison of applying techniques. The tables are organized same as Table 6.12.

Figure 7.5 shows the total number of edges versus the time to generate the model for FSMApp, ESG, and Crawler-based approach. Figure 7.5 shows the time increases

slowly compare to the other two approaches until 20 minutes with 150 edges. Then, the time reaches 60 minutes of FSMAApp because the behavior of the app is different, and learning the app functions for the first time needs more time. The ESG and the Crawler-based approach increase linearly above 200 edges. In general, since the tester's performance in the model build is measured, learning effects can occur. We can exclude point 60 minutes with 200 edges for FSMAApp then we will have a linear line for from 20 minutes to 30 minutes. Overall, Figure 7.5 shows FSMAApp scalable compared to the other two approaches in phase 1. Also, FSMAApp tests all apps with less time and edges using clusters.

Figure 7.6 shows the total number of test sequences versus the time to generate the test sequences. FSMAApp has a maximum of 29 test sequences with maximum time 50 minutes. The number of sequences has a bigger increase than FSMAApp because of the number of the nodes is high, whereas FSMAApp applies clusters. Also, the Crawler-based approach increases linearly because the approach does not support the loops in the model.

Figure 7.3 shows the total number of test inputs and actions versus the input selection time. The time increase linearly for less than 180 inputs and actions. It is increasing faster after 320 inputs more than ESG and the Crawler-based approach. We do not have any mobile apps with more than 500 inputs and actions. ESG and the Crawler-based approach has more than 500 inputs and action, and the time increase linearly. The other two approaches have a lot of input and actions because the number of test sequence is higher than FSMAApp.

Figure 7.4 shows the test LOC versus the execution/validation time. The execution/validation time increases rapidly with more than 1500 LOC for FSMAApp and ESG. The Crawler-based approach increases linearly. The approaches have a high increase. ESG and Crawling-based approach increases linearly. The largest app



shows that FSMApp has lowest LOC because of the number of inputs and actions, and the number of test steps.

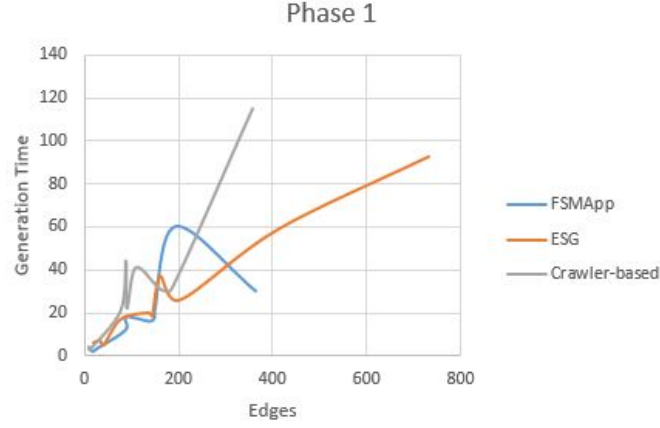


Figure 7.5: Generation Time vs number of Edges

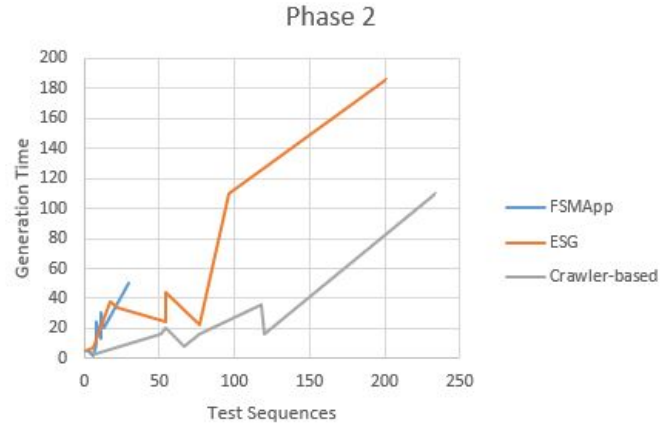


Figure 7.6: Generation Time vs Number of Test Sequences

### 7.7.5.3 RQ5.3: Efficiency

In this subsection, we compare the efficiency of FSMApp with ESG and Crawler-based approach. The efficiency is evaluated for all phases of test generation and execution.

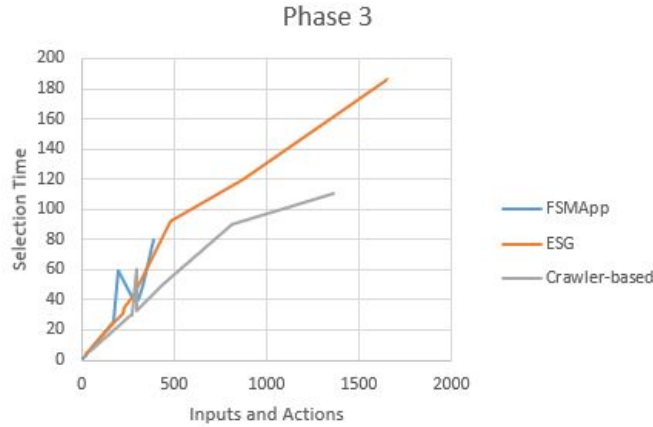


Figure 7.7: Input Selection Time vs Inputs and Actions

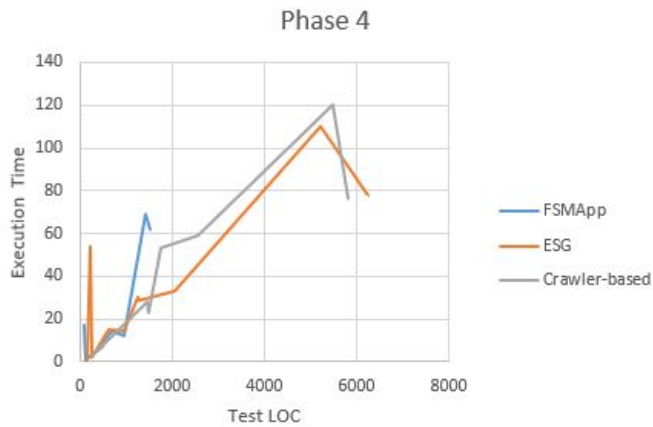


Figure 7.8: Execution/validation Time vs Test LOC

Table 7.14 shows the comparison of applying techniques in Game App. The FSMApp model is almost half of ESG, whereas we cannot apply the Crawler-based approach. Building the model for FSM takes much less time than ESG (3 versus 6 minutes), respectively. The reasons for this is the model for FSMApp is much smaller (6 nodes and 8 edges) than the model for ESG (12 nodes and 20 edges). The ESG model has twice the number nodes and more than double the edges compared to FSMApp. FSMApp generated 6 test sequences with eight steps, and the generation time is 2 minutes for Memory game app, as shown in Table 7.14. ESG generated

one test sequence with 39 steps, and the generated time is 5 minutes. There is a big difference in the number of test steps between FSMApp and ESG. For FSMApp, the total time for the four phases is 23 minutes. The FSMApp takes less than half the time ESG.

Table 7.9 shows the comparison result for Timber app. We built the model with FSMApp in 12 minutes, whereas it takes almost 20 minutes to build the model with ESG and Crawler-based approach. The difference is due to the clusters in FSMApp. The use of clusters reduces repeated of nodes and edges. FSMApp has almost 40% less than ESG and Crawler-based approach. FSMApp tests the Timber app with eight tests sequences compared to 77 and 66 test sequences for ESG and the Crawler-based approach, respectively as shown in Table 7.9. ESG generates a large number of test sequences because there are many loops in the model. The FSMApp has fewer test steps compared with other two approaches (120 versus 592 and 289 respectively). The ESG and the Crawler-based approach have far more steps because these approaches result in many repeated actions and have many more test sequences. As before, Crawler-based approach takes the least time to generate test sequences time. The overall time for testing is 66 versus 125 minutes and 88 minutes.

Table 7.11 shows the results of applying the techniques to the File Manager App. FSMApp reduces the model generation time and the model size by 70% compare to the ESG and Crawler-based approach. FSMApp uses 18 clusters to reduce the number of nodes and edges. The FSMApp model has 188 nodes, and 194 edges, while the Crawler-based approach model has 358 nodes and 357 edges. The ESG model is even bigger (476 nodes and 732 edges). For the File Manager app (Table 7.11), FSMApp generated few test sequences (29 versus 201 and 234, sequentially). The FSMApp has fewer steps compared with the other two approaches (381 versus

1451 and 1126, respectively). The time to generate test cases is roughly half of other approaches. The overall time for testing is 259 versus 489 and 435 minutes.

Table 7.10 reports the results of applying the techniques to the ML Manager. The table shows different results than for the other apps. The FSMApp model is a little bigger than ESG and far smaller than the Crawler-based approach. Building the model for FSMApp takes more time than ESG (by one minute). The reason for this result is that FSMApp has dummy nodes and edges between clusters. Also, to build the model for FSMApp takes less than half the time than the Crawler-based approach. The ML Manager app can be tested by 8 test sequences versus 54 and 77, respectively, for other two approaches. FSMApp needs less than half the steps of other two approaches, partly due to the reduction step test. FSMApp also need fewer inputs and actions. The overall time is 65 versus 131 and 124 minutes, respectively.

Table 7.7 shows the results of applying the techniques to the Simple Calendar. Building a model for FSMApp and ESG takes the same time, but both models take much less time (21 versus 27 minutes). The model for FSMApp has three more nodes than Crawler-based approach, whereas it has 5 fewer nodes than ESG. Also, The model for FSMApp has more edges than Crawler-based approach by 100%, whereas it is less than ESG by 13 edges. The difference comes from dummy nodes and edges. Simple Calendar can be tested by 8 test sequences versus 22 and 54 respectively for the other two approaches. The number of test sequences for FSMApp is very small compared to ESG because ESG has many loops. To makeup for the lack of loops in the Crawler-based approach, it needs to generate many more test sequences than both FSMApp and ESG. FSMApp considers components as inputs resulting in fewer edges in the model. A radio button is considered an input in FSMApp, but it is considered a node in the Crawler-based approach. FSMApp

has half steps of the other approaches because of the reduction step. There are also fewer inputs and actions. The overall test time is 88 versus 128 and 94 minutes.

Table 7.8 shows the results for the Amaze File Manager app model. The results shows similar results to the other apps. The FSMAApp model is more efficient and has fewer nodes and edges than the ESG and the Crawler-based approach. FSMAApp generated fewer test sequences (11 versus 54 and 120, respectively) . The FSMAApp also has fewer test steps compared to the other two approaches (188 versus 384 and 500 respectively). The time to generate test sequences is about half of the two other approaches. The overall time for testing is 83 versus 110 and 156 minutes. The time is almost double because of the large number of test sequences for the other two approaches.

Table 7.15 shows the comparison of applying the techniques to the Todo list app. Todo list app is the smallest app of the selected apps. the FSMAApp model is smaller than the ESG model and the Crawler-based approach. Building the model for FSMAApp takes larger than Crawler-based approach by two minutes. The FSMAApp model has more nodes and edges because of the clusters. FSMAApp generated 2 test sequences with 17 steps, the test generation sequence time is 5 minutes, as shown in Table 7.15. ESG generated 6 test sequences with 39 steps, and the rest sequence generation time is 6 minutes. The Crawler-based approach generated 7 test sequences and 19 test steps. There is a large difference in test steps between FSMAApp and ESG. However, the number of test steps in the Crawler-based approach is comparable. FSMAApp's total test time of the four phases is 11 minutes versus 18 and 12 minutes. The test time for FSMAApp is near the crawler-based test time because the test sequences are short and the app is small.

Table 7.12 shows the comparison of applying the techniques to the Minimal ToDo app. Building FSMAApp and Crawler-based approach models take the same time but

they both take more time. The FSMApp model has more nodes and edges because of the dummy nodes and edges. FSMApp generated 3 test sequences with 20 test steps, with a test sequence generation time of 5 minutes, as shown in Table 7.12. ESG generated 6 test sequences with 58 test steps, with generated time is 7 minutes. The Crawler-based approach generated 6 test sequences and 20 test steps. There is a large difference in test steps between FSMApp and ESG. The overall test time is 16 versus 25 and 19 minutes. The FSMApp performs much better than ESG but is close to the crawler-based approach because the test sequences are short and the app is small.

Table 7.13 shows a comparison of Applying the techniques in MIRAKEL Test Management. Building FSMApp and Crawler-based approach models have the same time, whereas both of them takes more time. The FSMApp model has more nodes and edges. The MIRAKEL app can be tested using FSMApp with 13 test sequences versus 96 and 118 respectively for the other two approaches. Also, the number of inputs and actions is much smaller. The overall test time is 151 versus 267 and 208 minutes. The difference shows from the fact that the other two approaches need more test steps and input. FSMApp saves time between 25% and 66% compare with ESG whereas FSMApp saves time between 6% and 48% compare than Crawler-based approach.

#### **7.7.5.4 RQ5.4: Effectiveness**

The case study executes the test cases and captures the number of defects. Tables 7.7 to 7.15 show the number of defect for FSMApp, ESG and Crawler-based approach in Phase 4. FSMApp, ESG, and Crawler-based approach found one defect. However, an additional ESG test failed because when the test performs a press back button action to the previous state, the test setup failed, because the

app exits instead of going to the previous page. Also for ESG, one test failed in Tables 7.7, 7.9 and 7.13 because of it executes a press back button.

Overall, FSMAApp compares favorably in effectiveness to the other two approaches. However, the high quality of the apps used in the case study make conclusions related to efficiency limited

## 7.8 Threats to Validity

We performed a number of the case studies to evaluate the applicability, scalability, effectiveness, and efficiency of FSMAApp to test Android mobile applications versus two other approaches, i.e. [88, 37].

We cannot yet generalize the results for other platforms. We only applied FSMAApp to Android Applications and did not consider IOS and Windows. The second issue is the configuration of automation tools which test a mobile application for one Android device only. The third issue is the knowledge of the functionality of the tested mobile apps and how the functions are linked. Our choice of apps that have high rating make effectiveness conclusions limited. We should follow up with less robust apps.

We already compared the FSMAApp results to ESG approach and the Crawler-based approach. The cost of execution is calculated as a function of the number of steps in the test sequences. This may not be appropriate in all cases, since some steps, with more inputs to enter, and longer App execution time may affect results. However, Chapter 4.1 successfully applied the same approach. Further, the number of nodes can be affected by developer experience when generating test paths for the FSMAApp, ESG or the Crawler-based approach Model.

Generalizability is limited as with any case study. We cannot guarantee that a

future case performs and gives the same result for other Android applications, for example, advanced games, reservation applications, and significant medical applications. We showed with our case studies that the FSMAApp could be applied to different categories of Android mobile application: a simple game, task management, file management, and music management.

Learning effects might bias the times needed to test each mobile app. One learning effect relates to the time it takes to understand how all functions in each mobile app work. If uncontrolled this could possibly lead to longer testing times for the first testing method applied to an app (generally this was FSMAApp). To avoid this confounding factor, we studied all functions for each of the 10 mobile apps in detail to understand all components and the connections between the mobile screens, before applying the testing approaches. These learning effects were thus controlled by carefully analyzing how apps work before applying any of the testing methods.

Exploratory testing is testing software without pre-designed test cases (based on some systematic method) [34]. We are not interested in exploratory testing. Input constraints effectively partition the input space into input values that will cause a desired transition or event vs. those that do not. Often, there are multiple values that fulfill any given input constraint. We leave it up to the tester to select among those. This leaves the possibility (explored by Hamlet et al. [112]) that some selected inputs that meet the constraints will uncover a fault, but others may not. As this is the case for all methods studied, they all face the same issue. We tried to select similar values, when possible, to mitigate this problem.



Table 7.3: Apps Components

Components	Simple Calen- dar	Amaze	Timber	ML Man- ager	File Man- ager	Minimal ToDo	MIRAKEL	Family Medicines List	Memory Game	Todo List
Buttons	17	14	26	14	25	1	78	27	2	2
TextField	14	8	2	0	0	2	13	7	0	1
List	0	3	2	0	14	0	6	1	0	1
Combobox	0	0	0	0	0	0	2	0	0	0
Activity Pages	23	0	0	0	0	5	0	4	0	2
Checkbox	7	2	10	0	0	0	3	0	0	0
Radiobox	9	1	0	0	0	0	0	0	0	0
Bottom Sheets	2	1	1	1	5	1	3	0	0	0
Chips	0	0	0	0	0	0	0	12	0	0
Data Tables	1	0	0	0	0	0	1	0	0	0
Grid Lists	0	13	0	1	0	0	0	0	0	1
Menus	0	1	1	0	1	0	1	0	0	1
Progress & Activity	0	0	0	0	0	0	1	0	0	0
Tabs	0	0	1	0	0	0	0	0	0	0
Toolbars	2	1	1	1	1	0	0	0	0	0
Images	0	0	0	0	0	0	0	0	20	0
Dialogs	0	1	5	0	0	4	0	0	0	0
Switch	12	20	0	3	9	2	2	0	0	0
Color Picker	7	0	2	1	2	0	0	0	0	0
Slider	1	0	1	0	0	0	0	0	0	0
Date Picker	4	0	0	0	0	0	0	0	0	0

Table 7.7: Comparison of Applying Techniques in Simple Calendar

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	82	87	79
		Number of edges	149	162	77
		Number of clusters	5	0	0
		Generation Time	21	37	21
2	Generate Test Sequence	Number of test Sequences	8	22	54
		Total test steps	105	237	229
		Generation time	25	34	20
	Total time (1) + (2)		46	71	41
3	Input Selection	#inputs	30	32	24
		#actions	276	245	240
		Time to choose input	40	43	30
	Total time (1) + (2) + (3)		86	114	71
4	Execute Test Cases	Test lines of code	970	955	1214
		Execution time	12	14	23
		Number of failed tests	0	4	0
		Number of passed tests	8	18	54
		Number of defects	0	1	0
	Total time		88	128	94

Table 7.8: Comparison of Applying Techniques in Amaze File Manager

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	83	133	160
		Number of edges	139	204	160
		Number of clusters	11	0	0
		Generation Time	16	26	31
2	Generate Test Sequence	Number of test Sequences	11	54	120
		Total test steps	188	384	500
		Generation time	13	25	16
	Total time (1) + (2)		29	51	47
3	Input Selection	#inputs	14	8	15
		#actions	266	210	427
		Time to choose input	40	30	50
	Total time (1) + (2) + (3)		69	81	97
4	Execute Test Cases	Test lines of code	676	1272	2558
		Execution time	14	29	59
		Number of failed tests	0	0	0
		Number of passed tests	11	46	120
		Number of defects	0	0	0
	Total time		83	110	156

Table 7.9: Comparison of Applying Techniques in Timber

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	52	85	90
		Number of edges	87	136	90
		Number of clusters	8	0	0
		Generation Time	12	20	22
2	Generate Test Sequence	Number of test Sequences	8	77	66
		Total test steps	120	592	289
		Generation time	9	22	8
	Total time (1) + (2)		21	42	30
3	Input Selection	#inputs	3	3	3
		#actions	190	475	295
		Time to choose input	59	92	60
	Total time (1) + (2) + (3)		59	92	60
4	Execute Test Cases	Test lines of code	483	2060	1444
		Execution time	7	33	28
		Number of failed tests	0	1	0
		Number of passed tests	8	76	66
		Number of defects	0	1	0
	Total time		66	125	88

Table 7.10: Comparison of Applying Techniques in ML Manager

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	87	78	109
		Number of edges	149	129	109
		Number of clusters	7	0	0
		Generation Time	18	17	41
2	Generate Test Sequence	Number of test Sequences	8	54	77
		Total test steps	122	256	304
		Generation time	11	44	16
	Total time (1) + (2)		29	61	57
3	Input Selection	#inputs	2	3	3
		#actions	170	228	298
		Time to choose input	26	35	33
	Total time (1) + (2) + (3)		55	96	90
4	Execute Test Cases	Test lines of code	523	1269	1753
		Execution time	10	30	34
		Number of failed tests	0	0	0
		Number of passed tests	8	54	77
		Number of defects	0	0	0
	Total time		65	131	124

Table 7.11: Comparison of Applying Techniques in File Manager

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	118	476	358
		Number of edges	194	732	357
		Number of clusters	18	0	0
		Generation Time	60	93	115
2	Generate Test Sequence	Number of test Sequences	29	201	234
		Total test steps	381	1451	1126
		Generation time	50	186	110
	Total time (1) + (2)		110	279	225
3	Input Selection	#inputs	11	12	11
		#actions	381	1111	1022
		Time to choose input	80	100	90
	Total time (1) + (2) + (3)		190	379	315
4	Execute Test Cases	Test lines of code	1430	5215	5466
		Execution time	69	110	120
		Number of failed tests	0	0	0
		Number of passed tests	29	201	234
		Number of defects	0	0	0
	Total time		259	489	435

Table 7.12: Comparison of Applying Techniques in Minimal ToDo

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	15	28	13
		Number of edges	23	42	12
		Number of clusters	2	0	0
		Generation Time	3	5	3
2	Generate Test Sequence	Number of test Sequences	3	6	6
		Total test steps	20	58	20
		Generation time	5	7	4
	Total time (1) + (2)		8	12	7
3	Input Selection	#inputs	6	7	9
		#actions	35	41	40
		Time to choose input	6	7	6
	Total time (1) + (2) + (3)		14	19	13
4	Execute Test Cases	Test lines of code	202	250	245
		Execution time	2	2	3
		Number of failed tests	0	0	0
		Number of passed tests	3	6	6
		Number of defects	0	0	0
	Total time		16	25	19

Table 7.13: Comparison of Applying Techniques in MIRAKEL: Task Management

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	167	245	186
		Number of edges	365	412	185
		Number of clusters	12	0	0
		Generation Time	30	59	31
2	Generate Test Sequence	Number of test Sequences	13	96	118
		Total test steps	277	778	630
		Generation time	20	110	36
	Total time (1) + (2)		50	169	67
3	Input Selection	#inputs	19	39	33
		#actions	308	836	781
		Time to choose input	49	120	90
	Total time (1) + (2) + (3)		89	189	126
4	Execute Test Cases	Test lines of code	1522	6256	5825
		Execution time	62	78	76
		Number of failed tests	0	1	0
		Number of passed tests	13	95	188
	Total time		0	0	0
			151	267	239



Table 7.14: Comparison of Applying Techniques in Memory Game Application

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	6	12	N/A
		Number of edges	8	20	N/A
		Number of clusters	0	0	N/A
		Generation Time	3	6	N/A
2	Generate Test Sequence	Number of test Sequences	6	1	N/A
		Total test steps	8	39	N/A
		Generation time	2	5	N/A
Total time (1) + (2)			5	11	N/A
3	Input Selection	#inputs	0	0	N/A
		#actions	5	19	N/A
		Time to choose input	1	3	N/A
Total time (1) + (2) + (3)			6	14	N/A
4	Execute Test Cases	Test lines of code	90	210	N/A
		Execution time	17	54	N/A
		Number of failed tests	0	0	N/A
		Number of passed tests	6	1	N/A
		Number of defects	0	0	N/A
Total time			23	68	N/A

Table 7.15: Comparison of Applying Techniques in Todo List

Step Id	Phase	Unit of Comparison	FSMApp	ESG	Crawler-based
1	Generate Model	Number of nodes	12	22	10
		Number of edges	18	34	9
		Number of clusters	1	0	0
		Generation Time	2	7	4
2	Generate Test Sequence	Number of test Sequences	2	3	7
		Total test steps	17	39	19
		Generation time	5	6	3
	Total time (1) + (2)		7	13	7
3	Input Selection	#inputs	3	3	3
		#actions	16	19	21
		Time to choose input	3	4	3
	Total time (1) + (2) + (3)		10	17	10
4	Execute Test Cases	Test lines of code	135	160	196
		Execution time	1	1	2
		Number of failed tests	0	0	0
		Number of passed tests	2	3	7
	Total time		11	18	12

## Chapter 8

### Future Work

In future work, the dissertation could be extended in many ways:

#### 8.1 Regression Testing

We need to extend FSMAApp for regression testing with case studies so that we can show applicability, scalability, and efficiency. Also, we would like to compare the approach with other regression testing approaches [37, 81, 94, 126].

#### 8.2 New system domains

This dissertation is focused on web and mobile applications domain. We plan to apply the FSMAApp in different system domains such as safety-critical systems, flight control systems, robotics device, IOT, and medical systems.

#### 8.3 Building Tools

We will build tools to automate model and test phases. This would decrease the cost of generating the model, test sequences, choose the input and action, write the

code for Appium, and execute them. Now, there is no tool to generate FSMApp model or test sequences.

## 8.4 Effectiveness

We would like to study the effectiveness of FSMApp with apply more case studies on pre-released apps.

## 8.5 Improving Efficiency of Test Execution

### 8.5.1 Efficiency Improvements During Execution

In this section, we explore ways how we can shorten execution of tests generated as test paths which are then turned into subtract tests and executable tests from FSMWeb in Subsection 3.2. Figure 8.1 shows an example graph used to illustrate our improvements. Figure 8.1 has five FSMs and three levels of hierarchy [45]. Table 8.1 shows the test paths through the graph that meet edge coverage.

#### 8.5.1.1 Serial Execution

During serial execution, test inputs are executed sequentially. Let  $t_i$  be a test,  $T$  be a set of tests  $T = t_1, \dots, t_k$ . If  $E(t)$  is the time required to set up and execute a test  $t$ , the total execution time for the set of tests  $T$  will be  $E_{serial}(T) = \sum_{i=1}^k E(t_i)$ .

In our example, we would execute the tests generated from the test paths in Table 8.1 one after another. Assuming the same effort as reported in [70] the test suite  $T$  would take three hours to execute.

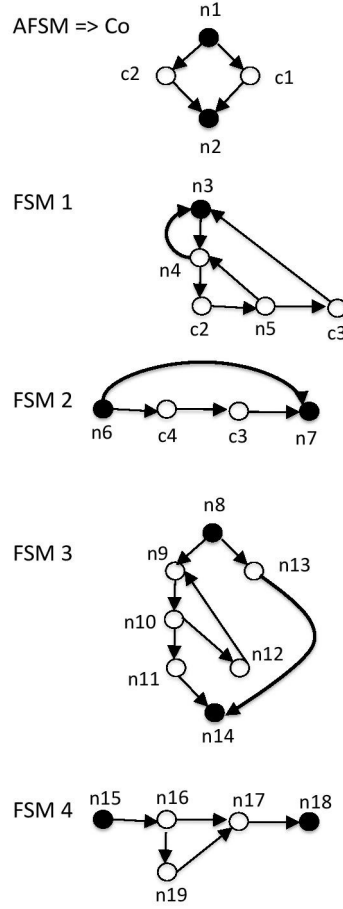


Figure 8.1: FSMs Example

#### 8.5.1.2 Parallel Launch

Launching test generation tasks in parallel can reduce  $E(T)$  by spreading the load among multiple processors. Then the time for a full parallel launch is  $E_{parallel}(T) = E(t_{max}) + S$ , where  $t_{max}$  is the execution time of the longest task and  $S$  is the time needed to divide the test suite among multiple threads or processes.  $S$  depends on, among other variables, the implementation, the speed and number of processors in the hardware, and the load of the system at the time of path execution.

Executing test generation tasks in parallel requires that the initial state be copied. Each task must maintain and operate only on its own copy of the sys-

Table 8.1: Test Paths for AFSM

$t_{01} : n_1 c_2 n_2$			
Test	Test Paths	Derivation Rules	Length
1	$n_1 t_{21} n_2$	$c_2 \rightarrow t_{21}$	4
2	$n_1 n_6 t_{41} t_{31} n_7 n_2$	$c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, \text{ and } c_3 \rightarrow t_{31}$	$11 = 4 + 4 + 3$
3	$n_1 n_6 t_{42} t_{31} n_7 n_2$	$c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, \text{ and } c_3 \rightarrow t_{31}$	$12 = 4 + 5 + 3$
4	$n_1 n_6 t_{41} t_{32} n_7 n_2$	$c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, \text{ and } c_3 \rightarrow t_{32}$	$16 = 4 + 4 + 8$
5	$n_1 n_6 t_{42} t_{32} n_7 n_2$	$c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, \text{ and } c_3 \rightarrow t_{32}$	$17 = 4 + 5 + 8$
$t_{02} = n_1 c_1 n_2$			
Test	Test Paths	Derivation Rules	Length
6	$n_1 n_3 n_4 t_{21} n_5 n_4 n_3 n_2$	$c_1 \rightarrow t_{11}, t_{11} \rightarrow n_3 n_4 c_2 n_5 n_4 n_3, \text{ and } c_2 \rightarrow t_{21}$	$9 = 7 + 2$
7	$n_1 n_3 n_4 n_6 t_{41} t_{31} n_7 n_5 n_4 n_3 n_2$	$c_1 \rightarrow t_{11}, t_{11} \rightarrow n_3 n_4 c_2 n_5 n_4 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, \text{ and } c_3 \rightarrow t_{31}$	$16 = 9 + 4 + 3$
8	$n_1 n_3 n_4 n_6 t_{42} t_{31} n_7 n_5 n_4 n_3 n_2$	$c_1 \rightarrow t_{11}, t_{11} \rightarrow n_3 n_4 c_2 n_5 n_4 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, \text{ and } c_3 \rightarrow t_{31}$	$17 = 9 + 5 + 3$
9	$n_1 n_3 n_4 n_6 t_{41} t_{32} n_7 n_5 n_4 n_3 n_2$	$c_1 \rightarrow t_{11}, t_{11} \rightarrow n_3 n_4 c_2 n_5 n_4 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, \text{ and } c_3 \rightarrow t_{32}$	$21 = 9 + 4 + 8$
10	$n_1 n_3 n_4 n_6 t_{42} t_{32} n_7 n_5 n_4 n_3 n_2$	$c_1 \rightarrow t_{11}, t_{11} \rightarrow n_3 n_4 c_2 n_5 n_4 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, \text{ and } c_3 \rightarrow t_{32}$	$22 = 9 + 5 + 8$
11	$n_1 n_3 n_4 t_{21} n_5 t_{31} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{21}, \text{ and } c_3 \rightarrow t_{31}$	$11 = 6 + 2 + 3$
12	$n_1 n_3 n_4 t_{21} n_5 t_{32} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{21}, \text{ and } c_3 \rightarrow t_{32}$	$16 = 6 + 2 + 8$
13	$n_1 n_3 n_4 n_6 t_{41} t_{31} n_7 n_5 t_{31} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, c_3 \rightarrow t_{31} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{31} \text{ (second } c_3)$	$18 = 8 + 4 + 3 + 3$
14	$n_1 n_3 n_4 n_6 t_{42} t_{31} n_7 n_5 t_{31} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, c_3 \rightarrow t_{31} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{31} \text{ (second } c_3)$	$19 = 8 + 5 + 3 + 3$
15	$n_1 n_3 n_4 n_6 t_{41} t_{32} n_7 n_5 t_{31} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, c_3 \rightarrow t_{32} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{31} \text{ (second } c_3)$	$23 = 8 + 4 + 8 + 3$
16	$n_1 n_3 n_4 n_6 t_{42} t_{32} n_7 n_5 t_{31} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, c_3 \rightarrow t_{32} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{31} \text{ (second } c_3)$	$24 = 8 + 5 + 8 + 3$
17	$n_1 n_3 n_4 n_6 t_{41} t_{31} n_7 n_5 t_{32} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, c_3 \rightarrow t_{31} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{32} \text{ (second } c_3)$	$23 = 8 + 4 + 3 + 8$
18	$n_1 n_3 n_4 n_6 t_{42} t_{31} n_7 n_5 t_{32} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, c_3 \rightarrow t_{31} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{32} \text{ (second } c_3)$	$24 = 8 + 5 + 3 + 8$
19	$n_1 n_3 n_4 n_6 t_{41} t_{32} n_7 n_5 t_{32} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{41}, c_3 \rightarrow t_{32} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{32} \text{ (second } c_3)$	$28 = 8 + 4 + 8 + 8$
20	$n_1 n_3 n_4 n_6 t_{42} t_{32} n_7 n_5 t_{32} n_3 n_2$	$c_1 \rightarrow t_{12}, t_{12} \rightarrow n_3 n_4 c_2 n_5 c_3 n_3, c_2 \rightarrow t_{22}, t_{22} \rightarrow n_6 c_4 c_3 n_7, c_4 \rightarrow t_{42}, c_3 \rightarrow t_{32} \text{ (first } c_3), \text{ and } c_3 \rightarrow t_{32} \text{ (second } c_3)$	$29 = 8 + 5 + 8 + 8$

tem state. Because of this, parallel execution requires more total memory usage than serial execution.

Each test generation task is implemented as a Java task that can be executed in a single thread. A supply of worker threads is supplied via Java's ThreadPool implementation. The tasks are assigned to worker threads according to a queue. When a worker completes its task, it is returned to the ThreadPool to receive another task. This process continues until there are no tasks remaining in the queue.

The degree of parallel execution is dependent on the number of threads in the thread pool, which may be scarce (there are more tasks than threads) or abundant (there are at least as many threads as there are tasks). If the threads are abundant, execution will proceed in the same manner as the parallel launch. If threads are

scarce, execution time will be longer.

We assume that there are  $q$  processors ( $q > 1$ ). We will also order the tests in descending execution time ( $E(T_1) \geq E(t_2) \geq \dots \geq E(t_h)$ ). We can then assign test to process ones as follows: Assign  $t_1 \dots t_q$  to process one 1 -  $q$ . Assign the next test to the processor  $i$  such a way that overall time for the tests assigned to  $i$  is minimal among all processors.

For both the ideal parallel launch and this possible Java Thread implementation, the number of processors available used is the main determining factor of performance improvement. In the example Java implementation, it is possible to create a ThreadPool containing, for example, 100 Java Threads, but this does not mean that all 100 Threads will be executing concurrently on a computer with four processing cores. At most, only a part of the processing cores would be available to the Java Virtual Machine, the others being used by the operating system and other processes. For this reason, it may be that significant improvement may require using a dedicated parallelism apparatus such as OpenCL or by otherwise distributing the tasks among multiple physical platforms.

Duarte et al. [96] presents a GridUnit framework to distribute the execution of software tests automatically. GridUnit uses a computational grid to distribute execution of JUnit test suites. GridUnit provides resources universalization for its nodes, creating isolated environments for test execution then there is no test case contamination. The task level is the test case. By contrast, our task level is one or more steps in a test case. Their assumption is that test cases are independent and should be executed in no pre-determined order. By contrast, we have an order due to sequential dependencies of parts of test case. This approach does not work with us because of the dependencies of the parts of test cases.

There is an exists testing infrastructure that do the parallel launch to avoid test

cross-contamination. Kappler [131] presents the proposed approach by developing a graph-based technique that parallels test execution with dependencies of other parts of test case. In future work, We will investigate his cloud tool to describe his parallel execution approach which depends of the store test case parts to avoid re-running the previous parts of test cases.

### 8.5.1.3 Parallel Execution from Points of Commonality

Rationale: many test paths have parts in common. In the example, test cases all start at a web portal that requires user authentication. This means that, if we can execute the test once, for the partial path that the tests have in common and then launch parallel execution from the last point of commonality, we can save test execution time. For example, let  $T = \{t_1, t_2, t_3\}$ ,  $t_1 = n_1, n_2, n_3, n_8$ ,  $t_2 = n_1, n_2, n_4, n_5, n_8$  and  $t_3 = n_1, n_2, n_4, n_6, n_7, n_8$ .

Point of commonality  $pc(t_i, \dots, t_j)$ :

$$pc(t_1, t_2, t_3) = n_1, n_2$$

$$pc(t_2, t_3) = n_4$$

Execute to earliest point of commonality (in tree)

We will execute  $n_1$  and  $n_2$  then pass the result to  $n_3$  and  $n_4$  to execute all the three tests.

## 8.5.2 Implementation

Capture-replay is an automated functional web testing approaches, and regression testing supports it. Software testers use capture-replay tools to capture the actions of GUI movements. Many capture-replay tools available, for example, Selenium IDE [23], jRapture [203] and SolEx [25]. The tools allow software testers



to record, modify and replay the test cases suite. Furthermore, the tools support management of the inputs and run the test cases for web applications. The capture-replay will capture the previous results of the last state in point of commonality.

Ostrand et al. [179] implemented an experimental test development environment (TDE). TDE raises the effectiveness of test cases for GUI and it links test designer, a test design library, and test generation engine with the capture-replay tools. Leatta et al. [143] compare between two web testing approaches capture-replay and programmable web testing. Leatta et al. [143] showed that capture-replay involves low developments but higher maintenance effort than programmable web testing. The capture-replay helps to capture the result of each execution of each part. Then, the new part uses the database to execute.

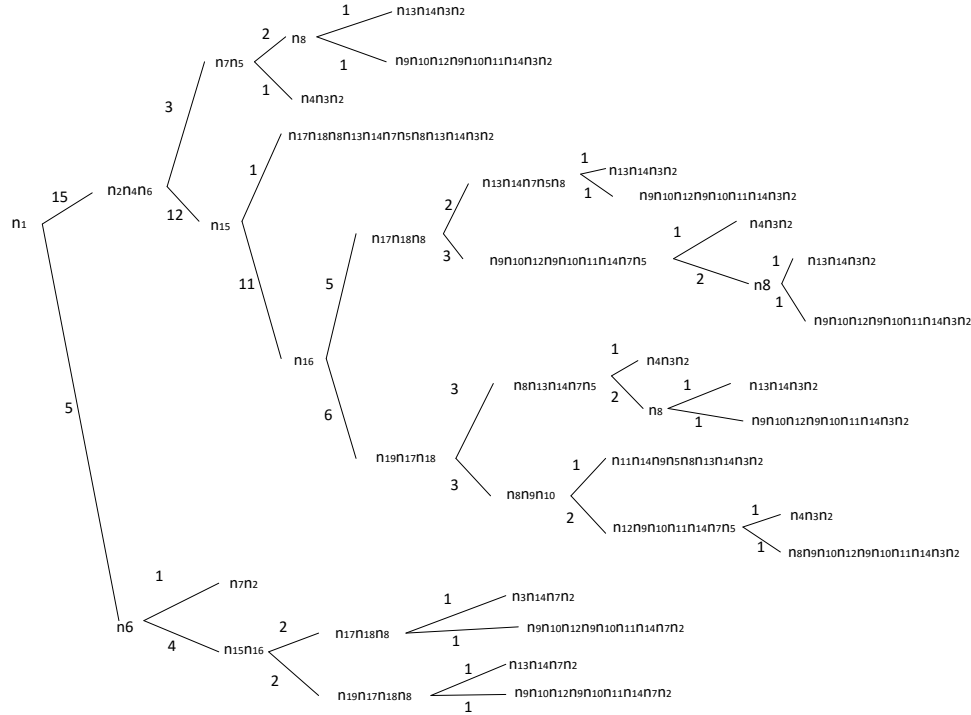


Figure 8.2: Commonality Subpaths

We applied the idea of last point of commonality on aggregated paths of table

8.1. Figure 8.2 shows the common sub paths between the paths. The numbers on the edge represent the number of execution.

Table 8.2: Save Execution

SubPath	Number of Node Executions	Number of Nodes	Total Nodes Executed
$n_8$	2	1	2
$n_7n_5$	3	2	6
$n_3n_4n_6$	15	3	45
$n_{15}$	12	1	12
$n_{16}$	11	1	11
$n_{17}n_{18}n_8$	5	3	15
$n_{13}n_{14}n_7n_5n_8$	2	5	10
$n_9n_{10}n_{12}n_9n_{10}n_{11}n_{14}n_7n_5$	3	9	27
$n_8$	2	1	2
$n_{19}n_{17}n_{18}$	6	3	18
$n_8n_{13}n_{14}n_7n_5$	3	5	15
$n_8$	2	1	2
$n_8n_9n_{10}$	3	3	9
$n_{12}n_9n_{10}n_{11}n_{14}n_7n_5$	2	7	14
$n_6$	6	1	6
$n_{15}n_{16}$	4	2	8
$n_{17}n_{18}n_8$	2	3	6
$n_{19}n_{17}n_{18}n_8$	2	4	8
single node	1	124	124
$n_1$	20	1	20
Total length of test paths	360		
Total length when applying Last point of commonality	180		
Saving (%)	50		

Table 8.2 presents the savings as a function of number of nodes executed when applying the last point of commonality. Column one shows the subpath in Figure 8.2. Column two shows the number of nodes to be executed for the test paths. Column

three shows the number of the nodes in the subpath and column four calculates the total of nodes executed.

## Chapter 9

### Conclusions

This dissertation is divided into two parts: Regression testing for web fail-safe applications and extension FSMWeb to test mobile apps. First, Testing mitigation of failures in the modified web applications is essential since defects can cause expensive outages, such as bank systems. We proposed a selective regression testing approach to test fail-safe behavior in web applications. Based on changes to the behavioral model, external failure types, and mitigation requirements (including failure applicability and weaving rules) it classified failure mitigation tests into retestable, reusable, and obsolete. It used partial regeneration for obsolete failure mitigation tests and generated new ones to achieve coverage. It used retestable as well as new failure scenarios. Reusable failure scenarios are removed when building new failure mitigation tests.

We compare test suite length for selective regression test versus a full retest for both an example and a large case study. Because the proportion of changes for the case study was small when compared to its overall size, selective regression testing was much more efficient. The overall selective regression testing effort is driven by how much change there is to the behavioral model (see also [45]) for a trade-off analysis related to changes to the FSMWeb behavioral model) and whether there

are multiple types of changes. The least expensive selective regression testing relates to changes in weaving rules. Finally, because potential failure scenarios tended to be small the genetic algorithm approach in [70] had to be abandoned, since it was less effective than coverage criteria ([69]). When changes to a web application are limited to specific functional areas and are not pervasive or affecting the whole system, the selective regression test approach presented can be expected to be more efficient than retesting the whole system.

Second, We extend the FSMWeb [51] to test the mobile application by compressing the model with the input components and reduction steps for test sequences. The FSMApp is a model-based black-box testing approach. FSMApp has four phases:

1. Generate Model: generate a hierarchical collection of FSMs model.
2. Generate Test Sequences: generate test sequences for each FSM then we aggregate them.
3. Input Selection: select the input constraints for the test sequence.
4. Execute Test Cases: execute the test sequences and run them by Appium.

We applied FSMApp on Family Medicine list app. The Family Medicine app has 45 nodes, 87 edges, and 9 clusters. By using our approach on this app, we generate the model, 11 test sequences with 87 steps, 48 inputs, 87 actions, and execute test cases with Appium. FSMApp has ten passed test cases, and one failed test case because the app was unable to change the patient and medicine name.

In this dissertation, we also define two approach ESG and the Crawler-based approach that also perform Black-Box MBT for Mobile Apps. de Cleva Farto et al. [88] used an Event Sequence Graph (ESG) to test mobile apps. Their approach consist of the following phases:

1. Create the Event Sequence Graph (ESG) test model. The ESG does not include (multiple) inputs in the graph explicitly rather, they are modeled with decision tables nodes.
2. Generate paths and implement test cases from the ESG model.
3. Execute implemented test cases with Robotium.

Amalfitano et al. [37] used a Crawler-based approach automated technique to test mobile applications. They create the model using a Crawler-based approach. The technique generates a GUI Tree using an iterative depth-first search and test cases are the events of the sequence from the root node of the tree to the leaf of the tree.

We apply FSMApp, ESG, and the Crawler-based approach on Family medicine list app. The FSMApp model is far smaller than the other two approaches in terms of model size. The FSMApp is smaller because the cluster required less repeated nodes and edges. Also, FSMApp generated 11 test sequences compared to 17 and 51 test sequences, respectively. The Crawler-based approach generates many test sequences because the approach does not support loops, and stops when it encounters a repeated screen. The FSMApp has significantly fewer total test steps (87 steps versus 219 and 240, respectively). The total number of inputs and actions is comparable for ESG and FSMApp, with 135 versus 150 inputs and actions. The Crawler-based approach requires a large number of inputs and actions because of repeated inputs and action.

ESG and the Crawler-based approach take 113 minutes, which is much higher than for FSMApp by 37 minutes. This is because model generation, time to choose inputs, and execution time for test cases required a shorter time than ESG and the Crawler-based approach. We can conclude that FSMApp is more efficient than ESG

and the Crawler-based approach.

This dissertation also proposed several case studies to investigate the applicability, scalability, efficiency, and effectiveness of FSMApp for testing mobile applications with ten mobile apps in different categories. Also, we compare FSMApp with ESG and the Crawler-based approach in these evaluation areas.

FSMApp can be applied to different app categories. FSMApp model is better than the other two approaches in 8 apps when comparing the test time and model size. ESG model for ML Manager app shows almost the same as FSMApp model for model building time and model size. Crawler-based approach model for the Simple calendar app has the same model building time as the FSMApp model, but the FSMApp is bigger than for the model size than Crawler-based approach.

FSMApp generates half of the test sequences compared with ESG and the crawler-based approach. The number of steps is almost half that of the other two approaches. Also, The execution time is much lower than ESG and the Crawler-based approach. FSMApp save between 25% and 66% of total time compare to ESG. Also, FSMApp save between 6% and 48% of total time compare to Crawler-based approach.

## Appendix A

### Android Terms

Table A.1: Android Terms

Terms	Description
Activity	An Activity is a page (screen) that is displayed to the application user. The Activity contains a set of layouts that organizes the item in the page.
View Widget	A View widget is GUI control that is in the layout of a mobile application such as label or EditText.
Service	A service is a background component to perform long-running tasks without user interaction such as playing video.
Broadcast Receiver	A Broadcast Receiver is an Android component to communication the application with Android system such as the battery is low.
Content Provider	A Content Provider manages the data on the database or the file system.
Intent Messaging	An Intent Messaging is a message object which is allowed to communicate between Activities, Services, and Broadcast Receivers.
Event Handler	The method handles the event from the event listener to execute the method registered.



## Appendix B

### Family Medicines List App Screens



Figure B.1: Family Medicines List App Screens

## Appendix C

### BNF Grammar for Input Selection Constraints

Table C.1 describes the grammar for input constraint language. Table C.1 extends the grammar language [51]. The input constraint contains single or multiple input choices, followed by none or any order constraints, followed by none or more propagation constraints. The input choices type can be required, optional or choice. The operators of required, optional and choice inputs are R, O, and C. R and O are followed by a list of string named inputs (parameters). They can have value like (Name = "Ahmed"). C is followed by an integer operator, such as C2, with optional bound (+ or -). Then, C followed by a list of named inputs. The integer operator shows the number of constraints selector. For example, C2 required to select precisely two input constraints. The choice input bound shows the limit of the selected constraints. For example, C2- is select input constraints up to 2 and C2+ is select input constraints at least 2.

The order of input constraints is sequential order (S) or any order (A). The order is following by an input constraint list. S order of input constraints should follow the order of the input list. For example, S(username, password, login) then the username entered first then password and finally click on login button. A order of input constraint should not follow any order. For example, A(name, date of birth)

then the inputs should be entered by any order name than the date of birth or date of birth then name. The input constraint list can include inputs or order constraints such as A(A(name, age, date), S(country, state, city)).

Propagation of input constraints can continue to use with other clusters. The continue operator followed by a list of input. The single operator is used to show that the input constraints cannot show again. The operators of Swap and switch for mobile application are W, and L. W is followed by a list input constraints such as W(Data Table), and L is followed a list of input constraints like L(logoff).

Table C.1: Grammar for Input Constraint Language

Constraint	(InputChoices [OrderConstraint] [PropagationConstraints]) — None
None	<b>none</b>
InputChoices	InputChoice [ InputChoice ]*
InputChoice	RequiredInput — OptionalInput — ChoiceInput
RequiredInput	Required InputList
Required	<b>R</b>
InputList	Left RequiredParameterList Right
Left	(
Right	)
RequiredParameterList	Parameter [ AdditionalParameter ]*
Parameter	SimpleParameter [ FixedConstraint ]
SimpleParameter	<b>string</b>
FixedConstraint	Equals Value
Equals	=
Value	Quate string Quote
Quote	'
AdditionalParameter	Comma Parameter
Comma	,
OptionalInput	Optional InputList
Optional	<b>O</b>
ChoiceInput	Choose Number [Bound] Left ParameterList Right
Choose	<b>C</b>
Bound	- — +
Number	<b>integer</b>
OrderConstraint	Order ConstraintList
ConstraintParameterList	ConstraintParameter [ AdditionalConstraintParameter ]*
ConstraintParameter	SimpleParameter — OrderConstraint
ConstraintList	Left ConstraintParameterList Right
AdditionalConstraintParameter	Comma ConstraintParameter
Order	AnyOrder — SequentialOrder
AnyOrder	<b>A</b>
SequentialOrder	<b>S</b>
PropagationConstraints	[ ContinueUseList ] [ SingleUseList ]
ContinueUseList	ContinueUse SimpleParameterList
ContineUse	<b>continue</b>
SingleUseList	SingleUse SimpleParameterList
SingleUse	<b>signle</b>
SimpleParameterList	SimpleParameter [ AdditionalSimpleParameter ]*
AdditionalSimpleParameter	Comma SimpleParameter
Single choice	<b>C1</b>
Multiple choice	<b>Cn</b>
Swipe	<b>w</b>
Scroll	<b>L</b>

## Appendix D

### Family Medicines List App

#### D.1 Clusters and Nodes

Table D.1: Clusters and Nodes For Main

Node	Cluster/LAP	Explanation
Main	Main Page	Main Screen of the mobile app
New Med	New Medication	Add new medication for the patient with his information
New Name	New Patient Name	Add new patient name with his information
Modify Med	Modify Medication	Edit medication information or delete the medication
Exit	Exit app	Exit the mobile application

Table D.2: Clusters and Nodes For New Med

Node	Cluster/LAP	Explanation
New Med	New Medication	Connection from main page
New medicines	New Medicines Information	Page related to medicines information
Dosage	Dosage Information	Page related to Dosage Information
Instruction	Instruction Information	Page related to Instruction Information
When	When Information	Page related to When Information
Add New	Add New	Submit new medicines information
Cancel	Cancel new	Cancel new medicines
Main Exit	To main page	Return to main page of mobile application

Table D.3: Clusters and Nodes For New Name

Node	Cluster/LAP	Explanation
New Name	New patient name	Connection from main page
New nameP	New patient name Information	Page related to patient name information
New medicines	New Medicines Information	Page related to medicines information
Dosage	Dosage Information	Page related to Dosage Information
Instruction	Instruction Information	Page related to Instruction Information
When	When Information	Page related to When Information
Add New	Add New	Submit new patient name information
Cancel	Cancel new	Cancel add new name patient
Main Exit	To main page	Return to main page of mobile application

Table D.4: Clusters and Nodes For Modify Med

Node	Cluster/LAP	Explanation
Modify Med	Modify Medicines	Connection from main page
Edit	Edit Medicines Information	Page related to edit information
Delete	Delete Medicines	Delete Medicines information
Cancel	Cancel	Cancel edit Medicines information

Table D.5: Clusters and Nodes For New NameP

Node	Cluster/LAP	Explanation
New NameP	Enter new name	Connection from Add new name page
Accept	Accept	Submit new patient name
Cancel	Cancel	Cancel new patient name

Table D.6: Clusters and Nodes For New Medicines

Node	Cluster/LAP	Explanation
New medicines	Enter new medicine	Connection from Add/Edit new medicine page
Accept	Accept	Submit new medicine
Cancel	Cancel	Cancel new medicine

Table D.7: Clusters and Nodes For Dosage

Node	Cluster/LAP	Explanation
Dosage	Enter new dosage	Connection from Add/Edit new medicine page
Accept	Accept	Submit new dosage
Cancel	Cancel	Cancel new dosage

Table D.8: Clusters and Nodes For When

Node	Cluster/LAP	Explanation
When	Enter new when	Connection from Add/Edit new medicine page
Accept	Accept	Submit new when
Cancel	Cancel	Cancel new when

Table D.9: Clusters and Nodes For Instructions

Node	Cluster/LAP	Explanation
Instructions	Enter new instruction	Connection from Add/Edit new medicine page
Accept	Accept	Submit new instruction
Cancel	Cancel	Cancel new instruction

Table D.10: Clusters and Nodes For Edit

Node	Cluster/LAP	Explanation
Edit	Edit medicine information	Connection from modifying page
Dosage	Dosage Information	Page related to Dosage Information
When	When Information	Page related to When Information
Instruction	Instruction Information	Page related to Instruction Information
Update	Update medicine	Submit medicine information
Cancel	Cancel	Cancel edit medicine

## D.2 Reduced Test Sequences and Test Values

Table D.11: Aggregated Test Paths Values

Id	Edge Id	Constraint	Values
1	A1	R(SelectN, buttonANM) S(SelectN, buttonANM) continue-use(SelectN)	selectN = "Trev" buttonANM = click
	D2	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	parMed = "Asprin" buttonANMA = click
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	A1	R(buttonANM)	buttonANM = click
Continued on next page			



**Table D.11 – continued from previous page**

<b>Id</b>	<b>Edge Id</b>	<b>Constraint</b>	<b>Values</b>
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	parMed = "Asprin" buttonANMA = click
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	H3	R(buttonAD)	buttonAD = click
	E1	R(parDosage, buttonADA) S(parDosage, buttonADA) Continue-use(parDosage)	parDosage = 25 mg buttonADA = click
	E2	O(parDosage), R(buttonADC) S(parDosage, buttonADC)	parDosage = 25 mg buttonADC = click
	B6	R(buttonAM)	buttonAM = click
	A4	R(buttonBack)	buttonBack = click
2	A1	R(SelectN, buttonANM) S(SelectN, buttonANM) continue-use(SelectN)	selectN = "Trev" buttonANM = click
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	parMed = "Asprin" buttonANMA = click
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	H5	R(buttonAW)	buttonAW = click
	G1	R(parWhen, buttonAWA) S(parWhen, buttonAWA) Continue-use(parWhen)	parWhen = "when needed" buttonAWA = click
	G2	O(parWhen), R(buttonAWC) S(parWhen, buttonAWC)	parWhen = "when needed" buttonAWC = click
	B5	R(buttonCNM)	buttonCNM = click
Continued on next page			

**Table D.11 – continued from previous page**

<b>Id</b>	<b>Edge Id</b>	<b>Constraint</b>	<b>Values</b>
	A4	R(buttonBack)	buttonBack = click
3	A1	R(SelectN, buttonANM) S(SelectN, buttonANM) continue-use(SelectN)	selectN = "Trev" buttonANM = click
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	parMed = "Asprin" buttonANMA = click
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	H5	R(buttonAI)	buttonAI = click
	G1	R(parInstruction, buttonAIA)  S(parInstruction, buttonAIA) Continue-use(parInstruction)	parInstruction = "when re- quired" buttonAIA = click
	G2	O(parInstruction), R(buttonAIC)  S(parInstruction, buttonAIC)	parInstruction = "when re- quired" buttonAIC = click
	B5	R(buttonCNM)	buttonCNM = click
	A4	R(buttonBack)	buttonBack = click
4	A2	R(buttonANN)	buttonANN = click
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	parMed = "Asprin" buttonANMA = click
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	H2	R(buttonANM)	buttonANM = click
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA)	parMed = "Asprin" buttonANMA = click
Continued on next page			

**Table D.11 – continued from previous page**

<b>Id</b>	<b>Edge Id</b>	<b>Constraint</b>	<b>Values</b>
		Continue-use(parMed)	
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	H1	R(buttonAD)	buttonAD = click
	E1	R(parDosage, buttonADA) S(parDosage, buttonADA) Continue-use(parDosage)	parDosage = 25 mg buttonADA = click
	E2	O(parDosage), R(buttonADC) S(parDosage, buttonADC)	parDosage = 25 mg buttonADC = click
	B6	R(buttonAnnA)	buttonAnnA = click
	A4	R(buttonBack)	buttonBack = click
5	A2	R(buttonANN)	buttonANN = click
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	parMed = "Asprin" buttonANMA = click
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	H3	R(buttonANNP)	buttonANNP = click
	E1	R(parName, buttonANNPA) S(parName, buttonANNPA) Continue-use(parName)	parName = "Trev" buttonANNPA = click
	C1	O(parName), R(buttonANNPC) S(parName, buttonANNPC)	parName = "Trev" buttonANNPC = click
	C2	R(buttonCNN)	buttonCNN = click
	A4	R(buttonBack)	buttonBack = click
6	A2	R(buttonANN)	buttonANN = click
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA)	parMed = "Asprin" buttonANMA = click
Continued on next page			

Table D.11 – continued from previous page

Id	Edge Id	Constraint	Values
		Continue-use(parMed)	
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	J2	R(buttonAW)	buttonAW = click
	F1	R(parWhen, buttonAWA) S(parWhen, buttonAWA) Continue-use(parWhen)	parWhen = "when needed" buttonAWA = click
	F2	O(parWhen), R(buttonAWC) S(parWhen, buttonAWC)	parWhen = "when needed" buttonAWC = click
	H6	R(buttonCNN)	buttonCNN = click
	A4	R(buttonBack)	buttonBack = click
7	A2	R(buttonANN)	buttonANN = click
	D1	R(parMed, buttonANMA) S(parMed, buttonANMA) Continue-use(parMed)	parMed = "Asprin" buttonANMA = click
	D2	O(parMed), R(buttonANMC) S(parMed, buttonANMC)	parMed = "Asprin" buttonANMC = click
	H5	R(buttonAI)	buttonAI = click
	G1	R(parInstruction, buttonAIA)  S(parInstruction, buttonAIA) Continue-use(parInstruction)	parInstruction = "when re- quired" buttonAIA = click
	G2	O(parInstruction), R(buttonAIC)  S(parInstruction, buttonAIC)	parInstruction = "when re- quired" buttonAIC = click
	H6	R(buttonCNN)	buttonCNN = click
	A4	R(buttonBack)	buttonBack = click
8	A3	O(SelectN), R(SelectM)	selectN = "Trev"
Continued on next page			

**Table D.11 – continued from previous page**

<b>Id</b>	<b>Edge Id</b>	<b>Constraint</b>	<b>Values</b>
		S(SelectN, SelectM) Continue-use(SelectN, SelectM)	selectM = "Asprin"
	I3	O(parName), R(parMed, buttonCE) S(parName, parMed, buttonCE)) continue-use(parName, parMed)	parName = "Trev" parMed = "Asprin" buttonCE = click
	I2	O(parName), R(parMed, buttonD) S(parName, parMed, buttonD) R(buttonD)	parName = "Trev" parMed = "Asprin" buttonD = click
	A4	R(buttonBack)	buttonBack = click
9	A3	O(SelectN), R(SelectM) S(SelectN, SelectM) Continue-use(SelectN, SelectM)	selectN = "Trev" selectM = "Asprin"
	I1	O(parName), R(parMed, buttonE) S(parName, parMed, buttonE) Continue-use(parName, parMed)	parName = "Trev" parMed = "Asprin" buttonE = click
	H3	R(buttonAD)	buttonAD = click
	E1	R(parDosage, buttonADA) S(parDosage, buttonADA) Continue-use(parDosage)	parDosage = 25 mg buttonADA = click
	E2	O(parDosage), R(buttonADC) S(parDosage, buttonADC)	parDosage = 25 mg buttonADC = click
	J5	R(buttonEEC)	buttonEEC = click
10	A4	R(buttonBack)	buttonBack = click
	A3	O(SelectN), R(SelectM) S(SelectN, SelectM) Continue-use(SelectN, SelectM)	selectN = "Trev" selectM = "Asprin"
	I1	O(parName), R(parMed, buttonE)	parName = "Trev"
Continued on next page			

**Table D.11 – continued from previous page**

<b>Id</b>	<b>Edge Id</b>	<b>Constraint</b>	<b>Values</b>
		S(parName, parMed, buttonE) Continue-use(parName, parMed)	parMed = "Asprin" buttonE = click
	H4	R(buttonAW)	buttonAW = click
	F1	R(parWhen, buttonAWA) S(parWhen, buttonAWA) Continue-use(parWhen)	parWhen = "when needed" buttonAWA = click
	F2	O(parWhen), R(buttonAWC) S(parWhen, buttonAWC)	parWhen = "when needed" buttonAWC = click
	J4	R(buttonEEU)	buttonEEU = click
	A4	R(buttonBack)	buttonBack = click
11	A3	O(SelectN), R(SelectM) S(SelectN, SelectM) Continue-use(SelectN, SelectM)	selectN = "Trev" selectM = "Asprin"
	I1	O(parName), R(parMed, buttonE) S(parName, parMed, buttonE) Continue-use(parName, parMed)	parName = "Trev" parMed = "Asprin" buttonE = click
	H5	R(buttonAI)	buttonAI = click
	G1	R(parInstruction, buttonAIA)  S(parInstruction, buttonAIA) Continue-use(parInstruction)	parInstruction = "When re- quired" buttonAIA = click
	G2	O(parInstruction), R(buttonAIC)  S(parInstruction, buttonAIC)	parInstruction = "When re- quired" buttonAIC = click
	J4	R(buttonEEU)	buttonEEU = click
	A4	R(buttonBack)	buttonBack = click

## D.3 Selenium Code of Test Paths

```

@Test
public void Test1() throws MalformedURLException {
    setUp();
    List

```

Figure D.1: Test 1 in Selenium

```

@Test
public void Test2() {
    MobileElement AddNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New Med\")");
    AddNewMed.click();
    MobileElement EnterNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter new Med\")");
    EnterNewMed.click();
    List<WebElement> textfeildsparMedicine = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parMedicine = textfeildsparMedicine.get(0);
    parMedicine.sendKeys("Med1");
    MobileElement EnterNewMedCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterNewMedCancel.click();
    EnterNewMed.click();
    parMedicine.sendKeys("Med1");
    MobileElement EnterNewMedAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterNewMedAccept.click();
    MobileElement EnterWhen = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter When\")");
    EnterWhen.click();
    List<WebElement> textfeildsWhen = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parWhen = textfeildsWhen.get(0);
    parWhen.sendKeys("When1");
    MobileElement EnterWhenAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterWhenAccept.click();
    MobileElement EnterWhen1 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"When1\")");
    EnterWhen1.click();
    parWhen.sendKeys("When1");
    MobileElement EnterWhenCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterWhenCancel.click();
    MobileElement AddNew = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New\")");
    AddNew.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    // exit
    wd.quit();
}

```

Figure D.2: Test 2 in Selenium



```

@Test
public void Test3() throws MalformedURLException {
    setUp();
    MobileElement AddNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New Med\")");
    AddNewMed.click();
    MobileElement EnterNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter new Med\")");
    EnterNewMed.click();
    List <WebElement> textfieldsparMedicine = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parMedicine = textfieldsparMedicine.get(0);
    parMedicine.sendKeys("Med2");
    MobileElement EnterNewMedCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterNewMedCancel.click();
    EnterNewMed.click();
    parMedicine.sendKeys("Med2");
    MobileElement EnterNewMedAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterNewMedAccept.click();

    MobileElement EnterInstruction = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter Instructions\")");
    EnterInstruction.click();
    List <WebElement> textfieldsInstruction = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parInstruction = textfieldsInstruction.get(0);
    parInstruction.sendKeys("Instruction2");
    MobileElement EnterInstructionAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterInstructionAccept.click();

    MobileElement EnterInstruction2 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Instruction2\")");
    EnterInstruction2.click();
    MobileElement cancelInstruction2 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    cancelInstruction2.click();

    MobileElement cancelAddMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    cancelAddMed.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.3: Test 3 in Selenium

```

@Test
public void Test4() throws MalformedURLException {

    setUp();
    MobileElement AddNewName = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New Name\")");
    AddNewName.click();
    MobileElement EnterNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter new Med\")");
    EnterNewMed.click();
    List <WebElement> textfeildsparMedicine = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parMedicine = textfeildsparMedicine.get(0);
    parMedicine.sendKeys("Med8");
    MobileElement EnterNewMedCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterNewMedCancel.click();
    EnterNewMed.click();
    parMedicine.sendKeys("Med8");
    MobileElement EnterNewMedAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterNewMedAccept.click();
    MobileElement EnterNewName = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click To Enter New Name\")");
    EnterNewName.click();
    List <WebElement> textfeildsNewName = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parNewName = textfeildsNewName.get(0);
    parNewName.sendKeys("NewName8");
    MobileElement EnterNewNameAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterNewNameAccept.click();
    MobileElement EnterNewName8 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"NewName8\")");
    EnterNewName8.click();
    parNewName.sendKeys("NewName8");
    MobileElement EnterNewNameCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterNewNameCancel.click();
    MobileElement AddNew = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New\")");
    AddNew.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.4: Test 4 in Selenium

```

@Test
public void Test5() throws MalformedURLException {
    setUp();
    MobileElement AddNewName = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New Name\")");
    AddNewName.click();
    MobileElement EnterNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter new Med\")");
    EnterNewMed.click();
    List<WebElement> textfeildsparMedicine = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parMedicine = textfeildsparMedicine.get(0);
    parMedicine.sendKeys("Med9");
    MobileElement EnterNewMedCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterNewMedCancel.click();
    EnterNewMed.click();
    parMedicine.sendKeys("Med9");
    MobileElement EnterNewMedAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterNewMedAccept.click();
    MobileElement EnterDosage = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter Dosage\")");
    EnterDosage.click();
    List<WebElement> textfeildsDosage = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parDosage = textfeildsDosage.get(0);
    parDosage.sendKeys("Dosage9");
    MobileElement EnterDosageAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterDosageAccept.click();
    MobileElement EnterDosage9 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Dosage9\")");
    EnterDosage9.click();
    parDosage.sendKeys("Dosage9");
    MobileElement EnterDosageCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterDosageCancel.click();
    MobileElement AddNew = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New\")");
    AddNew.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.5: Test 5 in Selenium



```

@Test
public void Test6() throws MalformedURLException {
    setUp();
    MobileElement AddNewName = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New Name\")");
    AddNewName.click();
    MobileElement EnterNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter new Med\")");
    EnterNewMed.click();
    List<WebElement> textfeildsparMedicine = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parMedicine = textfeildsparMedicine.get(0);
    parMedicine.sendKeys("Med11");
    MobileElement EnterNewMedCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterNewMedCancel.click();
    EnterNewMed.click();
    parMedicine.sendKeys("Med11");
    MobileElement EnterNewMedAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterNewMedAccept.click();
    MobileElement EnterWhen = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter When\")");
    EnterWhen.click();
    List<WebElement> textfeildsWhen = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parWhen = textfeildsWhen.get(0);
    parWhen.sendKeys("When11");
    MobileElement EnterWhenAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterWhenAccept.click();
    MobileElement EnterWhen11 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"When11\")");
    EnterWhen11.click();
    parWhen.sendKeys("When11");
    MobileElement EnterWhenCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterWhenCancel.click();
    MobileElement Cancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    Cancel.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.6: Test 6 in Selenium

```

@Test
public void Test7() throws MalformedURLException {
    setUp();
    MobileElement AddNewName = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New Name\")");
    AddNewName.click();
    MobileElement EnterNewMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter new Med\")");
    EnterNewMed.click();
    List<WebElement> textfieldsparMedicine = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parMedicine = textfieldsparMedicine.get(0);
    parMedicine.sendKeys("Med10");
    MobileElement EnterNewMedCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterNewMedCancel.click();
    EnterNewMed.click();
    parMedicine.sendKeys("Med10");
    MobileElement EnterNewMedAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterNewMedAccept.click();
    MobileElement EnterInstruction = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Click to enter Instructions\")");
    EnterInstruction.click();

    List<WebElement> textfieldsInstruction = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parInstruction = textfieldsInstruction.get(0);
    parInstruction.sendKeys("Instruction10");

    MobileElement EnterInstructionAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterInstructionAccept.click();
    MobileElement EnterInstruction10 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Instruction10\")");
    EnterInstruction10.click();
    parInstruction.sendKeys("Instruction10");
    MobileElement EnterInstructionCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterInstructionCancel.click();
    MobileElement AddNew = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Add New\")");
    AddNew.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.7: Test 7 in Selenium

```

@Test
public void Test9() throws MalformedURLException {
    setUp();
    List<WebElement> compoboxName = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.TextView\")");
    WebElement selectName = compoboxName.get(0);
    selectName.sendKeys("Trev");
    MobileElement EditMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Asprin : 25 mg\")");
    EditMed.click();
    List<WebElement> textfieldsEdit = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parName = textfieldsEdit.get(0);
    parName.clear();
    parName.sendKeys("Name5");
    WebElement parMedicine = textfieldsEdit.get(1);
    parMedicine.clear();
    parMedicine.sendKeys("Medicine5");
    MobileElement Edit = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Edit\")");
    Edit.click();
    MobileElement EnterDosage = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"25 mg\")");
    EnterDosage.click();
    List<WebElement> textfieldsDosage = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parDosage = textfieldsDosage.get(0);
    parDosage.sendKeys("Dosage5");
    MobileElement EnterDosageAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterDosageAccept.click();
    MobileElement EnterDosage5 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Dosage5\")");
    EnterDosage5.click();
    parDosage.sendKeys("Dosage5");
    MobileElement EnterDosageCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterDosageCancel.click();
    MobileElement Update = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Update\")");
    Update.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.8: Test 9 in Selenium



```

@Test
public void Test10() throws MalformedURLException {
    setUp();
    List<WebElement> compoboxName = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.TextView\")");
    WebElement selectName = compoboxName.get(0);
    selectName.sendKeys("Trev");
    MobileElement EditMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Asprin : 25 mg\")");
    EditMed.click();
    List<WebElement> textfeildsEdit = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parName = textfeildsEdit.get(0);
    parName.clear();
    parName.sendKeys("Trev");
    WebElement parMedicine = textfeildsEdit.get(1);
    parMedicine.clear();
    parMedicine.sendKeys("Asprin");
    MobileElement Edit = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Edit\")");
    Edit.click();
    MobileElement EnterWhen = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"When needed\")");
    EnterWhen.click();
    List<WebElement> textfeildsWhen = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parWhen = textfeildsWhen.get(0);
    parWhen.sendKeys("When7");
    MobileElement EnterWhenAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterWhenAccept.click();
    MobileElement EnterWhen7 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"When7\")");
    EnterWhen7.click();
    parWhen.sendKeys("When7");
    MobileElement EnterWhenCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterWhenCancel.click();
    MobileElement Cancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    Cancel.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.9: Test 10 in Selenium

```

@Test
public void Test11() throws MalformedURLException {

    setUp();

    List<WebElement> compoboxName = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.TextView\")");
    WebElement selectName = compoboxName.get(0);
    selectName.sendKeys("Trev");
    MobileElement EditMed = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Asprin : 25 mg\")");
    EditMed.click();

    List<WebElement> textfeildsEdit = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parName = textfeildsEdit.get(0);
    parName.clear();
    parName.sendKeys("Trev");
    WebElement parMedicine = textfeildsEdit.get(1);
    parMedicine.clear();
    parMedicine.sendKeys("Asprin");
    MobileElement Edit = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Edit\")");
    Edit.click();
    MobileElement EnterInstruction = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"When required\")");
    EnterInstruction.click();

    List<WebElement> textfeildsInstruction = wd.findElementsByAndroidUIAutomator("new UiSelector().className(\"android.widget.EditText\")");
    WebElement parInstruction = textfeildsInstruction.get(0);
    parInstruction.sendKeys("Instruction6");
    MobileElement EnterInstructionAccept = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Accept\")");
    EnterInstructionAccept.click();
    MobileElement EnterInstruction6 = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Instruction6\")");
    EnterInstruction6.click();
    parInstruction.sendKeys("Instruction6");
    MobileElement EnterInstructionCancel = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Cancel\")");
    EnterInstructionCancel.click();
    MobileElement Update = (MobileElement) wd
        .findElementByAndroidUIAutomator("new UiSelector().text(\"Update\")");
    Update.click();
    wd.pressKeyCode(AndroidKeyCode.BACK);
    wd.quit();
}

```

Figure D.10: Test 11 in Selenium



## Bibliography

- [1] Amaze file manager. <https://play.google.com/store/apps/details?id=com.amaze.filemanager>; <https://github.com/TeamAmaze/AmazeFileManager>. Accessed: 2018-02-25.
- [2] American psychological association (apa): The free on-line dictionary of computing. <http://dictionary.reference.com/browse/algorithmic testcase generation>. Accessed: February 15, 2015.
- [3] Android. <https://www.android.com/>. Accessed: 2018-10-17.
- [4] Anywhere software. <https://www.b4x.com/b4a.html>. Accessed: 2018-10-18.
- [5] App revenues. <https://www.businessofapps.com/data/app-revenues/>. Accessed: 2019-07-07.
- [6] Appbrain. <https://www.appbrain.com/stats/android-market-app-categories>. Accessed: 2019-01-14.
- [7] Appium. <http://appium.io/>. Accessed: 2016-01-26.
- [8] Eggplant. [http://developer.android.com/tools/help/monkeyrunner\\_concepts.html](http://developer.android.com/tools/help/monkeyrunner_concepts.html). Accessed: 2016-01-26.
- [9] Family medicines list. [http://basic384.rssing.com/chan-15081649/all\\_p2.html#item40](http://basic384.rssing.com/chan-15081649/all_p2.html#item40). Accessed: 2018-02-25.

- [10] File manager - storage, network, root manager. <https://play.google.com/store/apps/details?id=dev.dworks.apps.anexplorer>; <https://github.com/1hkr/AnExplorer>. Accessed: 2018-02-25.
- [11] Google play. <https://play.google.com/store>. Accessed: 2019-01-14.
- [12] Istqb® glossary of testing terms version: 2.4. <http://www.istqb.org/downloads/viewcategory/20.html>. Accessed: February 15, 2015.
- [13] Memory game application. <https://www.sourcecodester.com/android/8881/memory-game-application-android.html>. Accessed: 2018-02-25.
- [14] Minimal todo. <https://play.google.com/store/apps/details?id=com.avjindersinghsekhon.minimaltodo>; <https://github.com/avjinder/Minimal-Todo>. Accessed: 2018-02-25.
- [15] Mirakel: Task management. <https://play.google.com/store/apps/details?id=de.azapps.mirakelandroid&hl=en>; <https://github.com/MirakelX/mirakel-android>. Accessed: 2018-02-25.
- [16] ML manager. <https://play.google.com/store/apps/details?id=com.javiersantos.mlmanager>; <https://github.com/javiersantos/MLManager>. Accessed: 2018-02-25.
- [17] Monkeyrunner. [http://developer.android.com/tools/help/monkeyrunner\\_concepts.html](http://developer.android.com/tools/help/monkeyrunner_concepts.html). Accessed: 2016-01-26.
- [18] Null intent fuzzer. <http://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx>. Accessed: 2016-01-26.
- [19] Number of mobile app downloads worldwide in 2017, 2018 and 2022 (in billions). <https://www.statista.com/statistics/271644/>

- worldwide-free-and-paid-mobile-app-store-downloads/. Accessed: 2018-11-02.
- [20] Overview of conformance testing. <http://www.nist.gov/itl/ssd/is/overview.cfm>. Accessed: February 15, 2015.
- [21] Ranorex. <http://www.ranorex.com/>. Accessed: 2016-01-26.
- [22] Robotium. <https://github.com/robotiumtech/robotium>. Accessed: 2016-01-26.
- [23] Selenium. <http://www.seleniumhq.org/>. Accessed: 2018-01-20.
- [24] Simple calender. <https://play.google.com/store/apps/details?id=com.simplmobiletools.calendar>; <https://github.com/SimpleMobileTools/Simple-Calendar>. Accessed: 2018-02-25.
- [25] SolEx. <http://solex.sourceforge.net/>. Accessed: 2016-08-08.
- [26] Tau. <https://profilence.com/tau/>. Accessed: 2018-10-17.
- [27] Test ui for a single app. <https://developer.android.com/training/testing/ui-testing/espresso-testing>. Accessed: 2018-10-17.
- [28] Timber. <https://play.google.com/store/apps/details?id=naman14.timber>; <https://github.com/naman14/Timber>. Accessed: 2018-02-25.
- [29] Titanium mobile development environment. <https://www.appcelerator.com/Titanium/>. Accessed: 2016-01-26.
- [30] Todo list. <https://github.com/aaronksaunders/todolist.alloy>. Accessed: 2016-01-26.

- [31] Ui Automator. <http://developer.android.com/tools/testing-support-library/index.html>. Accessed: 2016-01-26.
- [32] Ui/application exerciser monkey. <https://developer.android.com/studio/test/monkey>. Accessed: 2018-10-16.
- [33] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 83–93. ACM, 2015.
- [34] Wasif Afzal, Ahmad Nauman Ghazi, Juha Itkonen, Richard Torkar, Anneliese Andrews, and Khurram Bhatti. An experiment on the effectiveness and efficiency of exploratory testing. *Empirical Software Engineering*, 20(3):844–878, 2015.
- [35] Maryam Ahmed, Rosziati Ibrahim, and Noraini Ibrahim. An adaptation model for android application testing with refactoring. *growth*, 9(10):65–74, 2015.
- [36] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, and Porfirio Tramontana. Agrippin: a novel search based testing technique for android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 5–12. ACM, 2015.
- [37] 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 (ICSTW)*, pages 252–261. IEEE, 2011.

- [38] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Nicola Amatuucci. Considering context events in event-based testing of mobile applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 126–133. IEEE, 2013.
- [39] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.*, pages 258–261. ACM, 2012.
- [40] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Ta, and Atif Memon. Mobiguitar—a tool for automated model-based testing of mobile apps. 2014.
- [41] 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.
- [42] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.
- [43] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.*, page 59. ACM, 2012.
- [44] A. Andrews, S. Boukhris, and S. Elakeili. Fail-safe testing of web applications. In *23rd Australian Software Engineering Conference (ASWEC)*, pages 200–209, April 2014.

- [45] A. 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 (HASE)*, pages 184–192, Jan 2014.
- [46] Anneliese Andrews, Ahmed Alhaddad, and Salah Boukhris. Black-box model-based regression testing of fail-safe behavior in web applications. *Journal of Systems and Software*, 149:318–339, 2019.
- [47] Anneliese Andrews, Salwa Elakeili, and Ahmed Alhaddad. Selective regression testing of safety-critical systems: a black box approach. In *Proceedings IEEE Conference on QRS, Workshop on Information Assurance.*, 2015.
- [48] Anneliese Andrews, Salwa Elakeili, and Salah Boukhris. Fail safe test generation in safety critical systems. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE)*.
- [49] Anneliese A. Andrews, S. Azghandi, and O. Pilskalns. Regression testing of web applications using FSMWeb. In *Proceedings IASTED International Conference on Software Engineering and Applications*, November 2010.
- [50] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery, Kshamta Jerath, and Roger Alexander. Scalability issues with using FSMWeb to test web applications. *Inf. Softw. Technol.*, 52(1):52–66, January 2010.
- [51] Anneliese Amschler Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. In *Software and System Modeling*, pages 326–345, 2005.

- [52] D. Ardagna, C. Cappiello, M.G. Fugini, E. Mussi, B. Pernici, and P. Plebani. Faults and recovery actions for self-healing web services. 15th International World Wide Web Conference., 2006.
- [53] Andrea Avancini and Mariano Ceccato. Security testing of the communication among android applications. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 57–63. IEEE, 2013.
- [54] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing.*, 1(1):11–33, 2004.
- [55] Tanzirul Azim and Iulian Neamtii. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.
- [56] Thomas Bäck. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In *Parallel Problem Solving from Nature 2, PPSN-II, Brussels, Belgium, September 28-30, 1992*, pages 87–96, 1992.
- [57] 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. ACM, 2016.
- [58] Florence Balagtas-Fernandez and Heinrich Hussmann. A methodology and framework to simplify usability analysis of mobile applications. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 520–524. IEEE Computer Society, 2009.

- [59] Ing Fevzi Belli. On the role of test sequence length, model refinement, and test coverage for reliability.
- [60] D. J. Berndt and A. Watkins. High volume software testing using genetic algorithms. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Volume 09*, pages 318–326, Washington, DC, USA, 2005. IEEE Computer Society.
- [61] Enrico Bertini, Silvia Gabrielli, and Stephen Kimani. Appropriating and assessing heuristics for mobile computing. In *Proceedings of the working conference on Advanced visual interfaces*, pages 119–126. ACM, 2006.
- [62] Bettina Biel, Thomas Grill, and Volker Gruhn. Exploring the benefits of the combination of a software architecture analysis and a usability evaluation of a mobile application. *Journal of Systems and Software*, 83(11):2031–2044, 2010.
- [63] Marco Billi, Laura Burzagli, Tiziana Catarci, Giuseppe Santucci, Enrico Bertini, Francesco Gabbanini, and Enrico Palchetti. A unified methodology for the evaluation of accessibility and usability of mobile applications. *Universal Access in the Information Society*, 9(4):337–356, 2010.
- [64] R Binder. Testing object oriented software: Models, patterns, and tools, 1999.
- [65] Solveig Bjornestad, Bjornar Tessem, and Lars Nyre. Design and evaluation of a location-based mobile news reader. In *2011 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–4. IEEE, 2011.
- [66] Jiang Bo, Long Xiang, and Gao Xiaopeng. Mobiletest: A tool supporting automatic black box test for software on smart mobile devices. In *Proceedings of*



*the Second International Workshop on Automation of Software Test.*, page 8. IEEE Computer Society, 2007.

- [67] Magdalena Borys and Marek Milosz. Mobile application usability testing in quasi-real conditions. In *2015 8th International Conference on Human System Interactions (HSI)*, pages 381–387. IEEE, 2015.
- [68] Salah Boukhris. *Fail Safe Testing in Web Applications*. PhD Dissertation, Computer Science Department, University of Denver, August, 2015.
- [69] Salah Boukhris, Ahmed Alhaddad, and Anneliese Andrews. A comparison of strategies to generate test requirements for fail-safe behavior. *The 15th International Conference on Software Engineering Research and Practice*, pages 3–9, 2017.
- [70] Salah Boukhris, Anneliese Andrews, Ahmed Alhaddad, and Rinku Dewri. A case study of black box fail-safe testing in web applications. *Journal of Systems and Software*, 131:146–167, 2017.
- [71] Marco Brambilla, Stefano Ceri, Sara Comai, and Christina Tziviskou. Exception handling in workflow-driven web applications. In *Proceedings of the 14th international conference on World Wide Web.*, WWW '05, pages 170–179, New York, NY, USA, 2005. ACM.
- [72] L.C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on UML designs. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 252–261, 2002.
- [73] Lionel C. Briand, Jie Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings*

of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE '02, pages 43–50, New York, NY, USA, 2002. ACM.

- [74] 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.
- [75] Mabel Vazquez Briseno and Pierre Vincent. Observations on performance of client-server mobile applications. In *1st International Conference on Information Technology, 2008. IT 2008.*, pages 1–4. IEEE, 2008.
- [76] Bruno Cabral and Paulo Marques. Exception handling: a field study in Java and .NET. In *Proceedings of the 21st European Conference on Object-Oriented Programming.*, ECOOP’07, pages 151–175, Berlin, Heidelberg, 2007. Springer-Verlag.
- [77] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
- [78] Gerardo Canfora, Francesco Mercaldo, Corrado Aaron Visaggio, Mauro D’Angelo, Antonio Furno, and Carminantonio Manganelli. A case study of automating user experience-oriented performance testing on smartphones. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 66–69. IEEE, 2013.
- [79] Erick Cantú-Paz and David E. Goldberg. Are multiple runs of genetic algorithms better than one? In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence David Davis, Rajkumar Roy, Una-May O’Reilly, Hans-Georg Beyer,

Russell Standish, Graham Kendall, Stewart Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, MitchA. Potter, AlanC. Schultz, KathrynA. Dowsland, Natasha Jonoska, and Julian Miller, editors, *Genetic and Evolutionary Computation — GECCO 2003*, volume 2723 of *Lecture Notes in Computer Science*, pages 801–812. Springer Berlin Heidelberg, 2003.

- [80] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.
- [81] 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. IEEE, 2018.
- [82] 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. IBM Press, 2002.
- [83] Yanping Chen, Robert L Probert, and Hasan Ural. Regression test suite reduction using extended dependence analysis. In *Fourth International Workshop on Software Quality Assurance: in Conjunction with the 6th ESEC/FSE Joint Meeting.*, pages 62–69. ACM, 2007.
- [84] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.

- [85] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? *2015 30th IEEE/ACM International Conference on Automated Software Engineering Automated*, 2015.
- [86] Marcello Cinque, Domenico Cotroneo, Zbigniew Kalbarczyk, and Ravishanker K Iyer. How do mobile phones fail? a failure data analysis of symbian os smart phones. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN'07.*, pages 585–594. IEEE, 2007.
- [87] 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 (QUATIC)*, pages 66–74. IEEE, 2014.
- [88] Guilherme de Cleve 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.
- [89] Kenneth Alan De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, Ann Arbor, MI, USA, 1975.
- [90] Marco de Sá and Luís Carriço. Lessons from early stages design of mobile applications. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, pages 127–136. ACM, 2008.
- [91] Márcio Eduardo Delamaro, Auri Marcelo Rizzo Vincenzi, and José Carlos Maldonado. A strategy to perform coverage testing of mobile applications. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 118–124. ACM, 2006.

- [92] Arilo C Dias-Neto and Guilherme H Travassos. A picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges. *Advances in Computers*, 80:45–120, 2010.
- [93] Pedro A Diaz-Gomez. *Optimization of parameters for binary genetic algorithms*. ProQuest, 2007.
- [94] 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 (MOBILE-Soft)*, pages 27–28. IEEE, 2016.
- [95] Kinga Dobolyi and Westley Weimer. Harnessing web-based application similarities to aid in regression testing. In *Software Reliability Engineering, 2009. ISSRE’09. 20th International Symposium on*, pages 71–80. IEEE, 2009.
- [96] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patrícia Machado. Gridunit: software testing on the grid. In *Proceedings of the 28th international conference on Software engineering*, pages 779–782. ACM, 2006.
- [97] James Edmondson, Aniruddha Gokhale, and Sandeep Neema. Automating testing of service-oriented mobile applications with distributed knowledge and reasoning. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–4. IEEE, 2011.
- [98] 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.

- [99] Majlinda Fetaji, Zamir Dika, and Bekim Fetaji. Usability testing and evaluation of a mobile software solution: a case study. In *30th International Conference on Information Technology Interfaces, 2008. ITI 2008.*, pages 501–506. IEEE, 2008.
- [100] Derek Flood, Rachel Harrison, and Claudia Iacob. Lessons learned from evaluating the usability of mobile spreadsheet applications. In *International Conference on Human-Centred Software Engineering*, pages 315–322. Springer, 2012.
- [101] Dominik Franke, Corinna Elsemann, and Stefan Kowalewski. Reverse engineering and testing service life cycles of mobile platforms. In *2012 23rd International Workshop on Database and Expert Systems Applications (DEXA)*, pages 16–20. IEEE, 2012.
- [102] Dominik Franke, Stefan Kowalewski, Carsten Weise, and Nath Prakobkosol. Testing conformance of life cycle dependent properties of mobile applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 241–250. IEEE, 2012.
- [103] Boni García and Juan Carlos Dueñas. Automated functional testing based on the navigation of web applications. *arXiv preprint arXiv:1108.2357*, 2011.
- [104] Anne Geraci, Freny Katki, Louise McMonegal, Bennett Meyer, John Lane, Paul Wilson, Jane Radatz, Mary Yee, Hugh Porteous, and Fredrick Springsteel. *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries*. IEEE Press, 1991.

- [105] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
- [106] Mati Golani and Avigdor Gal. Flexible business process management using forward stepping and alternative paths. In WilM.P. Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management.*, volume 3649 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin Heidelberg, 2005.
- [107] Tobias Griebe, Marc Hesenius, and Volker Gruhn. Towards automated ui-tests for sensor-based mobile applications. In *International Conference on Intelligent Software Methodologies, Tools, and Techniques*, pages 3–17. Springer, 2015.
- [108] Chenkai Guo, Jing Xu, Hongji Yang, Ying Zeng, and Shuang Xing. An automated testing approach for inter-application security in android. In *Proceedings of the 9th International Workshop on Automation of Software Test*, pages 8–14. ACM, 2014.
- [109] Yuepu Guo and Sreedevi Sampath. Web application fault classification - an exploratory study. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.*, ESEM '08, pages 303–305, New York, NY, USA, 2008. ACM.
- [110] Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering.*, 26(10):943–958, 2000.

- [111] Hyung Kil Ham and Young Bom Park. Mobile application compatibility test system design for android fragmentation. In *International Conference on Advanced Software Engineering and Its Applications*, pages 314–320. Springer, 2011.
- [112] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 206–215. IEEE, 1988.
- [113] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM, 2014.
- [114] Roei Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128. ACM, 2015.
- [115] Cuixiong Hu and Iulian Neamtii. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test.*, pages 77–83. ACM, 2011.
- [116] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, page 18. ACM, 2014.
- [117] Yongjian Hu, Tanzirul Azim, and Iulian Neamtii. Versatile yet lightweight record-and-replay for android. In *ACM SIGPLAN Notices*, volume 50, pages 349–366. ACM, 2015.



- [118] Arief Ernst Hühn, Vassilis-Javed Khan, Andrés Lucero, and Paul Ketelaar. On the use of virtual environments for the evaluation of location-based applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2569–2578. ACM, 2012.
- [119] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [120] Muhammad Zohaib Z Iqbal, Zafar I Malik, Aamer Nadeem, et al. 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. ACM, 2007.
- [121] Antti Jaaskelainen, Mika Katara, Antti Kervinen, Mika Maunumaa, Tuula Paakkonen, Tommi Takala, and Heikki Virtanen. Automatic gui test generation for smartphone applications-an evaluation. In *ICSE-Companion 2009. 31st International Conference on Software Engineering-Companion Volume, 2009.*, pages 112–122. IEEE, 2009.
- [122] Antti Jääskeläinen, Antti Kervinen, and Mika Katara. Creating a test model library for gui testing of smartphone applications (short paper). In *The Eighth International Conference on Quality Software, 2008. QSIC'08.*, pages 276–282. IEEE, 2008.
- [123] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 Inter-*

- national Symposium on Software Testing and Analysis.*, pages 67–77. ACM, 2013.
- [124] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. Modeling and test case generation of inter-component communication in android. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 113–116. IEEE, 2015.
  - [125] Bo Jiang, Xiang Long, Xiaopeng Gao, Zhifang Liu, and WK Chan. Floma: Statistical fault localization for mobile embedded system. In *2011 3rd International Conference on Advanced Computer Control (ICACC)*, pages 396–400. IEEE, 2011.
  - [126] Bo Jiang, Yu Wu, Yongfei Zhang, Zhenyu Zhang, and WK Chan. Retestdroid: Towards safer regression test selection for android application. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 235–244. IEEE, 2018.
  - [127] Shujuan Jiang and Yuanpeng Jiang. An analysis approach for testing exception handling programs. *SIGPLAN Not.*, 42(4):3–8, April 2007.
  - [128] Yiming Jing, Gail-Joon Ahn, and Hongxin Hu. Model-based conformance testing for Android. In *Advances in Information and Computer Security*, pages 1–18. Springer, 2012.
  - [129] Ryan Johnson, Zhaohui Wang, Angelos Stavrou, and Jeff Voas. Exposing software security and availability risks for commercial mobile devices. In *Reliability and Maintainability Symposium (RAMS), 2013 Proceedings-Annual*, pages 1–7. IEEE, 2013.

- [130] Jouko Kaasila, Denzil Ferreira, Vassilis Kostakos, and Timo Ojala. Testdroid: automated remote ui testing on android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, page 28. ACM, 2012.
- [131] Sebastian Kappler. Finding and breaking test dependencies to speed up test execution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1136–1138. ACM, 2016.
- [132] Heejin Kim, Byoungju Choi, and W Eric Wong. Performance testing of mobile applications at the unit test level. In *Third IEEE International Conference on Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009.*, pages 171–180. IEEE, 2009.
- [133] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [134] Konstantin Knorr and David Aspinall. Security testing for android mhealth apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–8. IEEE, 2015.
- [135] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST).*, pages 1–10. IEEE, 2015.
- [136] B. Korel, L.H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *Proceedings. International Conference on Software Maintenance, 2002.*, pages 214–223, 2002.

- [137] Artur H Kronbauer, Celso AS Santos, and Vaninha Vieira. Smartphone applications usability evaluation: a hybrid model and its implementation. In *International Conference on Human-Centred Software Engineering*, pages 146–163. Springer, 2012.
- [138] David Chenho Kung, Chien-Hung Liu, and Pei Hsia. An object-oriented web test model for testing web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 111–120. IEEE, 2000.
- [139] D.C. Kung, Chien-Hung Liu, and P. Hsia. An object-oriented web test model for testing web applications. In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, pages 537–542.
- [140] Tomi Lämsä. Comparison of gui testing tools for android applications. 2017.
- [141] Phillip A. Laplante, editor. *Dictionary of Computer Science Engineering and Technology*. CRC Press, Inc., Boca Raton, FL, USA, 2001.
- [142] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-based vulnerability testing for web applications. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 445–452. IEEE, 2013.
- [143] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *WCRE*, pages 272–281, 2013.
- [144] Barbara Staudt Lerner, Stefan Christov, Leon J. Osterweil, Reda Bendraou, Udo Kannengiesser, and Alexander Wise. Exception handling patterns for

- process modeling. *IEEE Transactions on Software Engineering.*, 36(2):162–183, 2010.
- [145] Florian Lettner and Clemens Holzmann. Automated and unsupervised user interaction logging as basis for usability evaluation of mobile applications. In *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia*, pages 118–127. ACM, 2012.
- [146] Hareton KN Leung and Lee White. Insights into regression testing software. In *Proceedings. Conference on Software Maintenance, 1989.*, pages 60–69. IEEE, 1989.
- [147] H.K.N. Leung and L. White. A cost model to compare regression test strategies. In *Conference on Software Maintenance, 1991., Proceedings.* , pages 201–208, Oct 1991.
- [148] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 519–530. ACM, 2014.
- [149] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410. IEEE, 2017.
- [150] Zhi-fang Liu, Bin Liu, and Xiao-peng Gao. Soa based mobile application software test framework. In *8th International Conference on Reliability, Maintainability and Safety, 2009. ICRMS 2009.*, pages 765–769. IEEE, 2009.

- [151] Zhifang Liu, Xiaopeng Gao, and Xiang Long. Adaptive random testing of mobile application. In *2010 2nd International Conference on Computer Engineering and Technology (ICCET)*, volume 2, pages V2–297. IEEE, 2010.
- [152] Begoña Losada, Maite Urretavizcaya, Juan-Miguel López-Gil, and Isabel Fernández-Castro. Combining intermod agile methodology with usability engineering in a mobile application development. In *Proceedings of the 13th International Conference on Interacción Persona-Ordenador*, page 39. ACM, 2012.
- [153] 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 (WCSE)*, pages 37–40. IEEE, 2012.
- [154] Qin Lu, Weishi Zhang, Bo Su, and Xiuguo Zhang. Exception handling policies for composite web services and their formal description. In *2007. NPC Workshops. IFIP International Conference on Network and Parallel Computing Workshops.*, pages 793–798, Sept.
- [155] Yu Lu, Pan Zulie, Liu Jingju, and Shen Yi. Android malware detection technology based on improved bayesian classification. In *2013 Third International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC)*, pages 1338–1341. IEEE, 2013.
- [156] Li Ma and Jeff Tian. Analyzing errors and referral pairs to characterize common problems and improve web reliability. In JuanManuelCueva Lovelle, BernardoMartínGonzález Rodríguez, JoseEmilioLabra Gayo, María Puerto Paule Ruiz, and LuisJoyanes Aguilar, editors, *Web Engineering.*, volume 2722

of *Lecture Notes in Computer Science*, pages 314–323. Springer Berlin Heidelberg, 2003.

- [157] Aravind Machiry, Rohan Tahirani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.
- [158] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609. ACM, 2014.
- [159] Aaron Marback, Hyunsook Do, and Nathan Ehresmann. An effective regression testing approach for php web applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 221–230. IEEE, 2012.
- [160] 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.
- [161] Mona Masood and Menaga Thigambaram. The usability of mobile applications for pre-schoolers. *Procedia-Social and Behavioral Sciences*, 197:1818–1826, 2015.
- [162] Robert McNitt. Storage Technology System Testing Manager, Private Communication. 2001.

- [163] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.
- [164] Abel Méndez-Porras, Christian Quesada-López, and Marcelo Jenkins. Automated testing of mobile applications: A systematic map and review. pages 195–208, 2015.
- [165] Ali Mesbah and Arie Van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 210–220. IEEE, 2009.
- [166] 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.
- [167] Tiago Monteiro and Ana CR Paiva. Pattern based gui testing modeling environment. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 140–143. IEEE, 2013.
- [168] Inês Coimbra Morgado, Ana CR Paiva, and Joao Pascoal Faria. Automated pattern-based testing of mobile applications. In *2014 9th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 294–299. IEEE, 2014.
- [169] 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.



- [170] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.
- [171] W. van der Aalst N. Russell and A. ter Hofstede. Exception handling patterns in process-aware information systems. PM Center Report BPM-06-04, 2006.
- [172] Miguel Nabuco and Ana CR Paiva. Model-based test case generation for web applications. In *International Conference on Computational Science and Its Applications*, pages 248–262. Springer, 2014.
- [173] Miguel Nabuco, Ana CR Paiva, Rui Camacho, and Joao 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.
- [174] Miguel Nabuco, Ana CR Paiva, and João Pascoal Faria. Inferring user interface patterns from execution traces of web applications. In *International Conference on Computational Science and Its Applications*, pages 311–326. Springer, 2014.
- [175] Leckraj Nagowah and Gayeree Sowamber. A novel approach of automation testing on mobile devices. In *2012 International Conference on Computer & Information Science (ICCIS)*, volume 2, pages 924–930. IEEE, 2012.
- [176] 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. ACM, 2012.

- [177] Duc Hoai Nguyen, Paul Strooper, and Jörn Guy Süß. Automated functionality testing through guis. In *Proceedings of the Thirty-Third Australasian Conference on Computer Science-Volume 102*, pages 153–162. Australian Computer Society, Inc., 2010.
- [178] 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.
- [179] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. A visual test development environment for gui systems. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 82–92. ACM, 1998.
- [180] Tuomas Pajunen, Tommi Takala, and Mika Katara. Model-based testing with a general purpose keyword-driven test automation framework. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 242–251. IEEE, 2011.
- [181] Robert M. Patton, Annie S. Wu, and Gwendolyn H. Walton. A Genetic Algorithm Approach to Focused Software Usage Testing. In *Annals of Software Engineering*, pages 259—286. Springer, 2003.
- [182] Soila Pertet and Priya Narasimhan. Causes of failure in web applications. Technical report, Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-05-109, December 2005.
- [183] Tan Phat Pham, Khasfariyati Razikin, Dion Hoe-Lian Goh, Thi Nhu Quynh Kim, Huynh Nhu Hop Quach, Yin-Leng Theng, Alton YK Chua, and Ee-Peng Lim. Investigating the usability of a mobile location-based annotation

- system. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*, pages 313–320. ACM, 2010.
- [184] Tapani Puhakka and Marko Palola. Towards automating testing of communicational b3g applications. In *Proceedings of the 3rd international conference on Mobile technology, applications & systems*, page 27. ACM, 2006.
- [185] L. Ran, C. Dyerson, A. Andrews, R. Bryce, and Ch. Mallery. Building Test Cases and Oracles to Automate the Testing of Web Database Applications. *Information and Software Technology*, 51:460–477, 2009.
- [186] Lihua Ran, Curtis Dyreson, Anneliese Andrews, Renée Bryce, and Christopher Mallery. Building test cases and oracles to automate the testing of web database applications. *Inf. Softw. Technol.*, 51(2):460–477, February 2009.
- [187] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, volume 12, pages 107–120, 2012.
- [188] Hassan Reza, Kirk Ogaard, and Amarnath Malge. A model based testing technique to test web applications using statecharts. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 183–188. IEEE, 2008.
- [189] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.
- [190] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

- [191] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.
- [192] Caspar Ryan and Pablo Rossi. Software, performance and resource utilisation metrics for context-aware mobile applications. In *null*, page 12. IEEE, 2005.
- [193] 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.
- [194] Sébastien Salva and Stassia R Zafimiharisoa. Data vulnerability detection by security testing for android applications. In *Information Security for South Africa, 2013*, pages 1–8. IEEE, 2013.
- [195] Michele Sama, Sebastian Elbaum, Franco Raimondi, David S Rosenblum, and Zhimin Wang. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, 36(5):644–661, 2010.
- [196] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5. ACM, 2014.
- [197] E. Sawadpong, P. Allen and B. Williams. Exception handling defects: An empirical study. In *2012 IEEE International Symposium on High-Assurance Systems Engineering*, pages 90–97, 2012.

- [198] Ronny Seiger and Thomas Schlegel. Test modeling for context-aware ubiquitous applications with feature petri nets. In *Proceedings of the Workshop on Model-based Interactive Ubiquitous Systems (MODIQUITOUS)*, 2012.
- [199] Sakura She, Sasindran Sivapalan, and Ian Warren. Hermes: A tool for testing mobile device applications. In *Software Engineering Conference, 2009. ASWEC'09. Australian*, pages 121–130. IEEE, 2009.
- [200] Saurabh Sinha and Mary Jean Harrold. Analysis of programs with exception-handling constructs. In *icsm*, page 348. IEEE, 1998.
- [201] Hyungkeun Song, Seokmoon Ryoo, and Jin Hyung Kim. An integrated test automation framework for testing on heterogeneous mobile platforms. In *2011 First ACIS International Symposium on Software and Network Engineering*, pages 141–145. IEEE, 2011.
- [202] Oleksii Starov. Cloud platform for research crowdsourcing in mobile testing. 2013.
- [203] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. *jRapture: A capture/replay tool for observation-based testing*, volume 25. ACM, 2000.
- [204] Mark D Syer, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal.*, 23(3):485–508, 2014.
- [205] 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.

- [206] Chuanqi Tao and Jerry Gao. Modeling mobile application test platform and environment: testing criteria and complexity analysis. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pages 28–33. ACM, 2014.
- [207] Abbas Tarhini, Hacène Fouchal, and Nashat Mansour. Regression testing web services-based applications. *AICCSA*, 6:163–170, 2006.
- [208] R Tay, N Chavannes, P Futter, GH Ng, and N Kuster. Failure modes and effects analysis (fmea) on the rf performance of mobile device terminals. In *The European Conference on Antennas and Propagation: EuCAP 2006*, volume 626, 2006.
- [209] Scott Tilley and Tauhida Parveen. Migrating software testing to the cloud. In *2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–1. IEEE, 2010.
- [210] Rafael Tolosana-Calasanz, José A. Bañares, Pedro Álvarez, Joaquín Ezpeleta, and Omer F. Rana. Exception handling patterns for hierarchical scientific workflows. In *Proceedings of the 6th International Workshop on Middleware for Grid Computing.*, MGC '08, pages 10:1–10:6, New York, NY, USA, 2008. ACM.
- [211] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-Based Testing Approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.
- [212] Heli Väättäjä and Anssi A Männistö. Bottlenecks, usability issues and development needs in creating and delivering news videos with smart phones. In

- Proceedings of the 3rd workshop on Mobile video delivery*, pages 45–50. ACM, 2010.
- [213] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.
  - [214] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.
  - [215] Vaninha Vieira, Konstantin Holl, and Michael Hassel. A context simulator as testing support for mobile apps. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 535–541. ACM, 2015.
  - [216] K Vijayalakshmi. Analysis of android os smart phones using failure mode and effect analysis. *International Journal of Latest Trends in Engineering and Technology*, 4(4):11–18, 2014.
  - [217] Liliana Vilela and Ana CR Paiva. Paradigm-cov: A multidimensional test coverage analysis tool. In *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–7. IEEE, 2014.
  - [218] Sergiy Vilkomir and Brandi Amstutz. Using combinatorial approaches for testing mobile applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 78–83. IEEE, 2014.
  - [219] Sergiy Vilkomir, Katherine Marszalkowski, Chauncey Perry, and Swetha Mahendrakar. Effectiveness of multi-device testing mobile applications. In *Mobile*

- Software Engineering and Systems (MOBILESoft)*, 2015 2nd ACM International Conference on, pages 44–47. IEEE, 2015.
- [220] Isabel Karina Villanes, Erick Alexandre Bezerra Costa, and Arilo Claudio Dias-Neto. Automated mobile testing as a service (am-taas). In *2015 IEEE World Congress on Services (SERVICES)*, pages 79–86. IEEE, 2015.
  - [221] Peng Wang, Bin Liang, Wei You, Jingzhe Li, and Wenchang Shi. Automatic android gui traversal with high coverage. In *2014 Fourth International Conference on Communication Systems and Network Technologies (CSNT)*., pages 1161–1166. IEEE, 2014.
  - [222] Zhimin Wang. *Validating context-aware applications*. The University of Nebraska-Lincoln, 2008.
  - [223] Zhimin Wang, Sebastian Elbaum, and David S Rosenblum. Automated generation of context-aware tests. In *Proceedings of the 29th international conference on Software Engineering*, pages 406–415. IEEE Computer Society, 2007.
  - [224] 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. ACM, 2010.
  - [225] Qunyi Wei, Zhaoxin Chang, and Qin Cheng. Usability study of the mobile library app: an example from chongqing university. *Library Hi Tech*, 33(3):340–355, 2015.
  - [226] Hsiang-Lin Wen, Chia-Hui Lin, Tzong-Han Hsieh, and Cheng-Zen Yang. Pats: A parallel gui testing framework for android applications. In *Computer Soft-*



- ware and Applications Conference (COMPSAC), 2015 IEEE 39th Annual, volume 2, pages 210–215. IEEE, 2015.
- [227] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
  - [228] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering.*, pages 250–265. Springer, 2013.
  - [229] Fan Ye and Tim Kelly. Component failure mitigation according to failure type. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, COMPSAC 2004.*, pages 258–264. IEEE, 2004.
  - [230] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 68. ACM, 2013.
  - [231] Lian Yu, Wei Tek Tsai, Yanbing Jiang, and Jerry Gao. Generating test cases for context-aware applications using bigraphs. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 137–146. IEEE, 2014.
  - [232] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, pages 183–192. IEEE, 2014.

- [233] 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.
- [234] Liangzhao Zeng, Hui Lei, Jun-Jang Jeng, J. Y Chung, and B. Benatallah. Policy-driven exception-management for composite web services. In *Seventh IEEE International Conference on E-Commerce Technology, 2005. CEC 2005.*, pages 355–363, July.
- [235] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):32, 2014.
- [236] Tao Zhang, Jerry Gao, Jing Cheng, and Tadahiro Uehara. Compatibility testing service for mobile applications. In *2015 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 179–186. IEEE, 2015.
- [237] Tao Zhang, Jerry Gao, Tadahiro Uehara, et al. Testing location-based function services for mobile applications. In *2015 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 308–314. IEEE, 2015.
- [238] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 214–224. ACM, 2015.
- [239] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards black box testing of android apps. In *2015 10th International Conference on Availability, Reliability and Security (ARES)*, pages 501–510. IEEE, 2015.

- [240] Liu Zhifang, Liu Bin, and Gao Xiaopeng. Test automation on mobile device. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 1–7. ACM, 2010.