

CAPITAL UNIVERSITY OF SCIENCE AND
TECHNOLOGY, ISLAMABAD



Energy Aware Test-Suite Prioritization for Android Applications

by

Maria Sagheer

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the

Faculty of Computing

Department of Computer Science

2021

Copyright © 2021 by Maria Sagheer

All rights reserved. No part of this thesis may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, by any information storage and retrieval system without the prior written permission of the author.

I dedicate my thesis to ALLAH ALMIGHTY, my parents, supervisor and friends who encouraged and supported me. A special feeling of gratitude for my parents for their endless love and support



CERTIFICATE OF APPROVAL

Energy Aware Test-Suite Prioritization for Android Applications

by

Maria Sagheer

(MCS191006)

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Basit Shahzad	NUML, Islamabad
(b)	Internal Examiner	Dr. Nadeem Anjum	CUST, Islamabad
(c)	Supervisor	Dr. Aamer Nadeem	CUST, Islamabad

Dr. Aamer Nadeem

Thesis Supervisor

April, 2021

Dr. Nayyer Masood
Head
Dept. of Computer Science
April, 2021

Dr. M. Abdul Qadir
Dean
Faculty of Computing
April, 2021

Author's Declaration

I, **Maria Sagheer** hereby state that my MS thesis titled “**Energy Aware Test-Suite Prioritization for Android Applications**” is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.

(**Maria Sagheer**)

Registration No: MCS191006

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled “**Energy Aware Test-Suite Prioritization for Android Applications**” is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.

(Maria Sagheer)

Registration No: MCS191006

Acknowledgement

All worship and praise is for ALLAH (S.W.T), the creator of whole worlds. I am obliged to Allah Almighty the Merciful, the Beneficent and the source of all Knowledge, for granting me the courage and knowledge to complete this thesis. He bestowed me with the patience, wellness and understanding to complete my thesis.

I am also thankful to my parents for their love support and sacrifices which they have made for me. Especially to my father, for the trust they have bestowed on me and to my mother who played an important role in keeping me motivated during the thesis.

A special thanks to Dr. Aamer Nadeem for his guidance, motivation and for helping me throughout in accomplishing my MS thesis. I am sincerely grateful to him for his continuous support during my research. His guidance, support and motivation enabled me in achieving the objectives of the project. He is the most respectable man and working under his supervision was an honor for me.

I also want to thanks Mr. Qamar Zaman for their comments and feedback on my research work. I am truly grateful how they keep me motivated during my research. I would also like to thank the CSD research group for their guidance. Finally, I would like to thank everyone who helped and supported me.

(Maria Sagheer)

Abstract

Rising popularity of smartphone applications has led us to the need for energy aware testing. Performing energy testing along with the functional testing is a labor-intensive task specially for those large sized applications that are continuously evolving. Software usually evolves in order to cope with the changing market or consumer needs. Currently there is a dearth of energy-based testing techniques that consider program's energy bugs in parallel with the functional bugs.

Regression testing (RT) is required when there is a need to test the modified part of the system. RT techniques provide confidence that the newly incorporated changes do not affect the unchanged parts of the system. RT is further classified into: TCP (Test Case Prioritization); which prioritizes the test suite based on some criteria, TSM (Test Suite Minimization); aims to eliminate the redundant test cases and TCS (Test Case Selection); that selects only those test cases which traverse the modified part of the system. In general, a number of techniques exist that perform TCP using traditional white box or black box coverage criteria. Likewise, some of the researchers presented techniques based on application's energy consumption and energy optimization. But hardly any of the research has been conducted on test case prioritization using the energy bugs and functional bugs as primary coverage criteria. The existing techniques in energy aware domain focus on test case minimization at application level by considering energy coverage as the major coverage criteria. These techniques calculate the energy cost of each segment and then perform optimization based on the energy greedy segments. While other techniques focus on only the multiple code coverage criterion.

In this work, we have proposed a TCP approach that uses code level energy bugs and statement coverage as primary criterion. The proposed approach uses the additional strategy to prioritize the test suite. The energy bugs coverage has been gathered by statically analyzing the code and then weights have been assigned to each test paths. The weights of each test path are calculated using the normalized energy-bug weight and the statement coverage. Using those assigned weights the

test paths are prioritized. Two variants of APFD metric (i.e., APFD-bug variant and APFD-Statement Coverage variant) have been used to evaluate the proposed technique. We have presented the evaluation results of 10 different Android applications. Results show that the proposed approach is able to detect 72-87% of the bugs.

Contents

Author’s Declaration	iv
Plagiarism Undertaking	v
Acknowledgement	vi
Abstract	vii
List of Figures	xi
List of Tables	xii
Abbreviations	xiv
1 Introduction	1
1.1 Regression Testing	1
1.2 Coverage-Based TCP	3
1.2.1 Code Coverage Criteria	4
1.2.2 Algorithm Based TCP	4
1.3 Energy Bugs and Hotspots	5
1.4 Purpose	7
1.5 Scope	7
1.6 Problem Statement	8
1.7 Research Methodology	8
1.8 Research Contribution	9
1.9 Thesis Structure	9
2 Literature Review	11
2.1 Test Case Prioritization	11
2.1.1 Code-Coverage Based TCP	12
2.1.2 Energy Based TCP	15
2.1.2.1 Energy Optimization	15
2.1.2.2 Energy Consumption	16
2.2 Analysis and Comparison	16
3 Proposed Approach	21

3.1	Proposed Solution	22
3.1.1	Prioritization Algorithm	24
3.1.1.1	Test Path's Weight Calculation	24
3.1.2	Evaluation	33
4	Implementation	34
4.1	Implementation Details	34
4.2	Usage Details	36
4.3	Mapping Example	38
4.3.1	Input Details	38
4.3.2	Output Details	39
5	Results and Discussion	42
5.1	Subject Programs	43
5.1.1	AndroidRun	43
5.1.2	Jamendo	43
5.1.3	OpenCamera	44
5.1.4	A2DPVolume	44
5.1.5	Apollo	44
5.1.6	Andromatic	44
5.1.7	K9Mail	45
5.1.8	MyTracks	45
5.1.9	Find3	45
5.1.10	WiFiGPSLocation	45
5.1.11	Features of Subject Program	45
5.2	Prioritization Example Mapping on Android-Run Application	46
5.2.1	onCreate():AndroidRun Application	47
5.2.2	onLoctaionChange():AndroidRun Application	53
5.3	Evaluation Parameters	56
5.3.1	APFD-Bug Variant	57
5.3.2	APFD-Statement Coverage Variant	58
5.4	Results	58
5.5	Comparison	60
6	Conclusion and Future Work	63
	Bibliography	65

List of Figures

3.1	Context Diagram of the Proposed Approach	22
3.2	Flowchart of Prioritization Algorithm	23
3.3	Example CFG	25
4.1	Class Diagram of Proposed Algorithm	35
4.2	File Format: Input Screen-I	37
4.3	File Format: Input Screen-II	37
4.4	Console Output Screen	37
4.5	File Format: Total Statement Coverage	38
4.6	File Format: Total Energy Bug Coverage	38
4.7	File Format: Test Path Energy Bug Coverage	39
4.8	File Format: Test Path Statement Coverage	39
4.9	Output: Prioritized List-I	40
4.10	Output: Prioritized List-II	40
4.11	Output: Prioritized List-III	41
4.12	Output: Prioritized List-IV	41
4.13	Output: Prioritized List-V	41
5.1	Resource Usage in Subject Programs	47
5.2	CFG of onCreate():AndroidRun Application	48
5.3	Test Paths of onCreate():AndroidRun Application	48
5.4	CFG of onLoctaionChange():AndroidRun Application	53
5.5	Test Paths of onLoctaionChange():AndroidRun Application	54
5.6	TCP Evaluation Metric Types [58]	57
5.7	Application's APFD _{Bug} , APFD _{StC} and APFD _(B+St) values	59

List of Tables

1.1	Classification of Energy Bugs [23]	6
1.2	Occurrence Scenario of Energy Bugs in Source Code	7
2.1	Literature Review: Code Based TCP	17
2.2	Literature Review: Energy Optimization	19
3.1	Test Paths from Example CFG	25
3.2	Representing assigned weight w.r.t energy bug occurrence	27
3.3	Resource Weights W.r.t Power Consumption	27
3.4	Energy-Bug Coverage Details:Example	29
3.5	Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-Step I:Example	31
3.6	Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-Step II:Example	32
3.7	Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-Step III:Example	32
5.1	TCP Resource, CFG Test Paths and Bug-prone paths in each Android applications	46
5.2	TCP Resource, CFG Test Paths and Bug-prone paths in each Android applications	49
5.3	Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step I	49
5.4	Additional Coverage Information of Test Paths of onCreate(): AndroidRun Application	50
5.5	Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step II	50
5.6	Additional Coverage Information of Test Paths of onCreate(): AndroidRun Application -II	51
5.7	Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step III	52
5.8	Additional Coverage Information of Test Paths of onCreate(): AndroidRun Application -III	52
5.9	Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step IV	52
5.10	Coverage Information of Test Paths of onLoctaionChange(): AndroidRun Application	54

5.11	onLocationChange(): Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step-I	55
5.12	onLocationChange(): Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-StepII	55
5.13	onLocationChange(): Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-StepIII	56
5.14	Application's $APFD_{Bug}$, $APFD_{StC}$ and $APFD_{(B+St)}$ values at $\alpha = 0.5$	58
5.15	Comparison of Proposed Approach with Existing Approach	60

Abbreviations

APFD	Average Fault Detection Percentage
API	Application Programming Interface
CFG	Control Flow Graph
GEMMA	Gui Energy Multi-objective optiMization for Android applications
GPS	Global Positioning System
GUI	Graphical User Interface
ILP	Integer Linear Programming
RT	Regression Testing
TCP	Test Case Prioritization
TSM	Test Suite Minimization
TSS	Test Suite Selection

Chapter 1

Introduction

Now-a-days software systems are continuously evolving due to the need of fixing detected bugs, incorporating additional functionalities and restructuring or re-designing system architecture. Therefore, the need for software testing arises to perform authenticating and evaluating the additional functionalities of software systems with the existing ones [1].

1.1 Regression Testing

Regression testing is performed for providing the confidence that the newly incorporated changes do not affect the previous functionality or unchanged part of the system [2]. Therefore, regression testing is required where there is need to test the newly built version of the software which is obtained by modifying previous version [3].

Regression testing is further categorized into: Corrective Regression testing and Progressive Regression testing. **Corrective regression** testing is the testing of the program which was obtained by making corrections to the previous version of the program while **Progressive regression** testing is testing the program which was obtained by adding the new functionalities in the previous program version [4].

As the software evolves, size of the test suite increases which makes it costly to execute the entire test suite [2]. Therefore, regression testing can be classified into three domains:

- Test case selection: aims to identify the test cases which traverse the modified and affected parts of the program. Selection technique recognizes those test cases which would be important to test the latest changes of software [5].
- Test suite minimization: aims at identifying and eliminating the redundant test cases with the goal to reduce the size of test suite. This test case reduction is based on some criteria for example if test cases t and t_1 belongs to the test suite T and covers the same function F of the program P , then any of the test case will be discarded in the favor of other. The selected test cases are placed in minimal hitting set, identifying which is an NP hard problem [6].

While, in test cases minimization the test cases are permanently eliminated. [3].

- Test case prioritization: aims to prioritize the test cases based on some criteria. This approach provides the ability to give priority to the highly significant test cases based on some measures like early fault detection [7].

Test case prioritization ranks the test cases using a suitable metric. Prioritization does not discard any test case rather it just prioritizes them. Regression testing involves execution of large number of test cases which may be time consuming. To cope with this problem the researchers have proposed various techniques for test case prioritization [8]. These approaches can be further classified into coverage-based TCP, historic information-based TCP, cost-aware, time-aware TCP and risk-aware TCP [9], further elaborated as follows.

1. Coverage-Based TCP: This is the fundamental approach in prioritization process which analyzes the code directly. Traditionally, branch, function and statements are considered as the most covered coverage criteria [10].

2. History-Based TCP: this technique uses the history information of each test case to prioritize the test cases. The technique aims to improve the fault detection rate by using the historic fault data of a test case [11].
3. Cost Cognizant TCP: Cost-aware prioritization technique was first presented by [12]. The proposed cost-cognizant technique estimates the number of faults detected per unit testing using genetic algorithm.
4. Time-aware TCP: Multi-objective time-aware prioritization aims to find the ideal sequence of the test cases within the given time constraint [13].
5. Risk-aware TCP: This technique is mainly applied in the projects which are concerned with the software's risk related values while developing. Due to this, some researchers use this information in the process of prioritization. Risk-aware prioritization aims to categorize the test cases which have to distinguish the system's risk fault in earlier stages [14].
6. Model-based TCP: This type of testing is applied where source code related information is not available. System models and output of the test cases are used to perform test case reordering. [15].

1.2 Coverage-Based TCP

Measuring coverage requires code portioning into sub-units like methods, branches or statements. Coverage based TCP approach chooses a level of partitioning and defines coverage over those elements [16].

In the field of test case prioritization the following two approaches are commonly used,

1. Code Coverage Criteria based approach
2. Algorithm based TCP

which are further explained as follows:

1.2.1 Code Coverage Criteria

Coverage based prioritization aims to provide maximum coverage of the program elements [9]. Broadly, white box and black box prioritization covers the coverage criteria-based TCP approaches.

Black box-based testing approaches do not require source code information. These types of approaches use software models and their interactions for testing purpose. System's input and outputs are used to prioritize the test cases according to the diversity measures [17]. T-wise, Input Model Diversity (IMD), Total Input Model Mutation etc. are the major black box TCP approaches.

Whereas, white box prioritization approaches require source code information making it more efficient in early detection of faults and provide complete coverage of the code. These type of prioritization approaches use additional or total strategy for providing the coverage to the basic code elements (statement, branch and method) [18].

1.2.2 Algorithm Based TCP

The traditional Test Case Prioritization algorithm is the greedy algorithm which chooses the locally optimal solution in every iteration to reach global optimal solution [3]. Researchers have worked in the area to propose approach that can produce the optimized solution. In literature, many algorithms have been proposed that use genetic programming or ILP based optimized prioritization approaches. Researchers have been working on nature inspired algorithms with the aim to increase the efficiency of achieving nearly optimal solution [19].

Conventionally white-box and black-box approaches have been used for test case prioritization [20]. Black box approaches do not require the code therefore preventing the need of source code availability. Conversely white box approaches

increase the source code coverage and hence increases the early fault detection [21]. The objective of existing prioritization techniques is to improve the fault detection rate for which APFD metric is used. However, energy consumption is the major concern in mobile applications [22]. Existing approaches focus on the fault detection but do not cover energy consumption which is of major concern in mobile applications. Similar to the conventional applications, mobile applications require functional correctness along with energy consumption. Therefore, the major aim of the study is to propose an energy-aware prioritization approach for Android applications.

1.3 Energy Bugs and Hotspots

Energy inefficiency is a state where malfunctioning applications may lead to inappropriate power states, such as energy greedy GPS/ background sensor updates, non-idle power state in absence of user activity and so on. Moreover, these inefficiencies appear when the application does not effectively utilize the device resources (i.e., not releasing resources like Wi-Fi/ GPS or expensive sensor updates) eventually hampering the battery life of smart devices.

Energy inefficiencies in smartphone applications can be broadly classified into energy hotspots and energy bugs described as:

Energy Hotspot: An energy hotspot can be elaborated as a scenario where application causes the device to abnormally consume high amount of power even if the resource utilization is very low.

Energy Bug: An energy bug can be described as a scenario where malfunctioning application restricts the device to be in idle state even when there is no user activity involved.

Table 1.1 shows the categories of energy bugs and energy hotspots that can be found in an Android application. These energy bugs will be covered in our proposed TCP technique.

TABLE 1.1: Classification of Energy Bugs [23]

Category	Energy Bug	Energy Hotspot
Hardware Resources	Resource Leak: Resources (such as the WiFi) that are acquired by an application during execution must be released before exiting or else they continue to be in a high-power state.	Suboptimal Resource Binding: Binding resources too early or releasing them too late causes them to be in high-power state longer than required.
Sleep-State transition heuristics	Wakelock Bug: Wakelock is a power management mechanism in Android through which applications can indicate that the device needs to stay awake. However, improper usage of Wakelocks can cause the device to be stuck in a high-power state even after the application has finished execution. This situation is referred to as a Wakelock bug.	Tail-Energy Hotspot: Network components tend to linger in a high-power state for a short-period of time after the workload imposed on them has completed. The energy consumed by the component between the period of time when the workload is finished and the component switches to the sleep-state is referred to as Tail Energy.
Background Services	Vacuous Background Services: In the scenario where an application initiates a service such as location updates or sensor updates but does not removes the service explicitly before exiting, the service keep on reporting data even though no application needs it.	Expensive Background Services: Background services such as sensor updates can be configured to operate at different sampling rates. Unnecessarily high sampling rate may cause energy hotspots and therefore should be avoided.
Defective Functionality	Immortality Bug: Buggy applications may re-spawn when they have been closed by the user, thereby continuing to consume energy.	Loop-Energy Hotspot: For instance, a loop containing network login code may be executed repeatedly due to reasons such as unreachable server.

TABLE 1.2: Occurrence Scenario of Energy Bugs in Source Code

Types	Bug occurrence scenario
A-Bug	Where any resource is acquired only but not used or released
AR-Bug	Where any resource is acquired and released but not used
AU-Bug	Where any resource is acquired and used but not released

In this study, we only work with the code level energy bugs. Whereas the table 1.2 shows the usage scenarios showing how bugs occur in the source code.

The proposed approach will prioritize the test cases on the basis of energy bugs where bug-prone path will be assigned higher priority as it will be able to detect the energy critical code segments.

1.4 Purpose

Energy is the major resource in the smartphones however the mobile application developers lack the objective information regarding the application behavior with respect to the energy bugs [24]. Therefore, the proposed approach will focus on identifying the energy critical paths (involving energy bug) which will be used as coverage criteria for TCP.

1.5 Scope

The study aims to prioritize the test suite in order to detect the energy critical code (i.e. code segments which may produce energy bugs) at early stages. For prioritizing the test suite, it is assumed that the test suite, application's source

code and related coverage information (including statement coverage and energy bug coverage) is available.

1.6 Problem Statement

Existing studies focus on code coverage criteria for performing regression testing on Android applications and is used for prioritizing or minimizing the test suite. But due to the increased usage of battery constrained devices, testing non-functional properties, especially energy bugs coverage, is recently gaining substantial interest. Therefore, energy bug coverage is the under-explored area in the domain of regression testing of Android applications. While, traditionally used code coverage metric is inadequate to provide energy bug coverage as it only uncovers functionality bugs. Considering code coverage criteria does not ensure the energy bug coverage of the code. whereas in literature, energy based minimization techniques are discussed which do not focus on the code coverage of the program.

1.7 Research Methodology

1. Initially, literature review has been done in order to find the most recent and relevant regression testing techniques specially TCP used for testing of Android applications. After studying various papers, we concluded that traditional coverage criteria are not enough for testing energy related aspects of smart devices. Therefore, the articles related to energy consumption in the domain of Android applications have been studied. Most of the studies focuses on method level coverage while using selective types of bugs.
2. To overcome the gap in the existing techniques, a new weight-based energy aware criterion has been proposed in order to find the energy-prone paths at early stages. The proposed test case prioritization approach calculates the weight of test paths based on energy bug coverage and the statement coverage.

3. Approach will be implemented as follows:
 - (a) Firstly, the source code of AUT (Application Under Test), test suite, its relative code coverage information and the test suite is assumed to be available.
 - (b) Weight of each test path is calculated considering the type of energy bug it contains as well as the statements it covers. We can variate by giving more preference to any of the given criteria (elaborated in Equation 2 Chapter 3).
 - (c) Based on the calculated weights the test paths will be added in the prioritized test suite. The algorithm uses the additional strategy for prioritizing the test suite.
 - (d) Then the proposed approach is then evaluated using the variants of APFD metric.
4. After generating the prioritized test suite, the evaluation will be performed using APFD-bug variant and statement variants (detailed explanation in chapter 5).

1.8 Research Contribution

In this research work, we have presented test case prioritization approach that uses energy bug and statement coverage as primary prioritization criteria. Previous TCP techniques aim to reveal the functional bugs but the proposed approach is able to uncover the functional bugs as well as the non-functional bugs (energy bug).

1.9 Thesis Structure

The thesis is organized into five different chapters where **Chapter 1** describes the introduction to the proposed approach and its objectives. **Chapter 2** explains the

literature review and presents the analysis on the existing techniques. **Chapter 3** discusses the proposed methodology and the generic example explaining the proposed solution. **Chapter 4** describes the implementation details. **Chapter 5** describes the experimental results and their comparison with other technique. In the end **Chapter 6** presents the conclusion and future work.

Chapter 2

Literature Review

Regression testing is a technique used for detecting the faults in software close to the ending phase of development life cycle. This testing technique is required to verify the additional functionalities of the system and to verify that previous system has not been affected after incorporating the changes. Test suite is developed in order to verify the system functionalities, therefore, its size tends to increase with the evolution of software over time. This increase in the size of test suite makes it expensive to execute in terms of time, effort and hence consumes around 80% of the whole budget. Due to the time, cost and resource constraints, it is needed to make regression testing more efficient in detecting faults. For early fault detection, regression testing has been categorized into three major sub-domains that are test suite minimization, test case selection and test case prioritization [25].

2.1 Test Case Prioritization

Test case prioritization reorganizes the test suite with the aim to increase the early fault detection rate and provides maximum benefits to the testers. While prioritizing the test suite, it is ensured that the high priority test cases should be executed earlier in the testing process [26]. Prioritization does not eliminate any

test case instead this approach uses a suitable metric to rank test cases in the test suite.

Rothermel et al, [5] presented the formal definition of test case prioritization as:

Definition: Test Case Prioritization Problem

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

Problem: To find

$$T' \in PT \text{ such that } (\forall T'' \in PT)(T' \succ T'')[f(T') \geq f(T'')].$$

Here PT, is the set of all possible prioritization permutations (ordering) of test suite T and f is the fitness function applied to yield award value for the ordering.

Several approaches for test suite prioritization exist in literature focusing on coverage criteria and prioritization algorithm. Likewise, energy aware approaches focus on measuring and estimating energy consumption. Based on the existing studies we have directed our research towards the coverage criteria-based test case prioritization and then energy estimation for Android applications. Therefore, the section is further subdivided into directions i.e., code-coverage based TCP and energy consumption of Android application.

2.1.1 Code-Coverage Based TCP

The existing papers in the category of code-coverage based TCP are briefly explained as follows:

In 2013, W. sun [8] proposed multi-objective test case prioritization strategy combining event and statement coverage. For the empirical study two popular GUI applications have been used to evaluate the fault detection capability of two strategies and reveals the inconsistencies between them.

In 2014, D. Hao [27] presented unified test case prioritization approach that constitutes both additional and total strategy. To evaluate the effectiveness of proposed approach 28 Java and 40 C programs were considered. Results demonstrate that a wide range of techniques derived using basic and extended models can be more effective than total techniques and competitive with additional techniques.

The approach presented in [17] is the extension of approach presented in 2016 [28] in which experimentation has been carried out on Java code having 12 lines. The result in [17] shows that 8 out 12 lines were covered by this approach showing 89.2% coverage.

In 2017, S. Wang [29] presented the quality aware TCP approach which make use of method and statement level code coverage. The proposed technique first inspects fault-prone code then calculate the weight of the code units and prioritize the test cases based on these weights. They have combined the dynamic method coverage with the total strategy and evaluate the results. Empirical evaluation has been conducted on 7 open-source Java applications (33 versions) shows that the proposed approach performed better as compared to others.

In 2018, IyadAlazzam [30] presented a TCP approach based on method and statement coverage aiming to increase the efficiency in error detection. Weight for individual test cases are calculated using average statement and method coverage for the system under test and then prioritize the test suite from high to low value of weight. This approach has been experimented on a Java application having 212 lines and 48 methods. But the approach is not compared with any of the traditional approaches.

In 2019, an empirical study has been conducted by N. M. Torres [31]. The focus of the study is to evaluate the prioritization approaches based on method, branch and coverage with APFD as well as with the m-spreading Matrix. Java programs have been used for evaluating the results showing that additional strategies perform better in terms of APFD value while techniques that follows total strategies performed better considering M-spreading values.

In 2019, T. Afzal et al. [32] presented path complexity and branch coverage-based TCP approach assuming that complex code is more likely to contain faults. The experiment has been carried out on three different Java programs and results has been compared with the traditional branch coverage criterion. The results show that proposed approach outperforms existing branch coverage-based approach in terms of APFD (Average Percentage of Faults Detected) up to 18% on average.

In 2018, A. Ammar et al. [33] presented weight-based TCP criteria with the objective to generate unique value for each test case. The proposed approach uses code coverage for prioritizing the test case in which weight is calculated based on five different criteria that are: statement, path, function, branch and fault coverage.

In case multiple weight of multiple test cases will be same then UniVal (unique Value) is calculate based on the type that whether test cases cover the same segments or they cover the different code segments. After adjusting the weights of test cases according to the type of code they cover, these test cases are then prioritized in descending order of their new weights. This proposed algorithm is evaluated on three different Java programs showing that enhanced weighted method performed well in all four code-coverage criteria except in functional coverage. [33]

In 2020, R. Huang et al. [34] proposed a new code combination coverage criterion for test case prioritization. This code coverage criteria combines the concepts of code coverage with combination coverage. An empirical study has been conducted on four different Java programs (14 versions) and five Unix applications (30 versions). To compute the effectiveness of proposed approach they presented a comparison of proposed approach with traditionally followed regression testing approaches: adaptive random, total, additional and search-based techniques.

In 2020, another statement-based defect prediction approach for TCP has been presented by Y. Shao et al.g [35]. There proposed approach first predicts the defects on code level and then use that information for prioritizing the test suite. The experimentation has been carried out on four different open-source datasets and results have been evaluated using APFD matrix.

2.1.2 Energy Based TCP

The gathered literature shows that the focus is on energy consumption and energy optimization therefore the code-based energy consumption can be further divided into two categories: energy optimization and energy consumption in smartphone applications.

2.1.2.1 Energy Optimization

In 2013, L. Ding et al [36] proposed a new TSM approach that encodes minimization problem as integer linear programming problem. The approach has been evaluated on two Android applications showing that while maintaining high code coverage energy consumption has been reduced to on average of 5 % to 48 %.

In 2014, D. Li et al [37] presented an extension to the previous technique [30] in a way to allow testers to generate energy efficient reduced test suite. The extended approach selects test cases based on software test requirements and their energy consumption. Evaluation result shows that the approach produces 95% energy efficient test suite in less than a second.

In 2016, R. Jabbarvand et al [38] presented energy aware TSM approach aims to reduce the size of test suite while effectively testing the energy properties of Android application. The approach calculates the eCoverage value of test cases based on energy greedy methods of the program. The proposed approach is then evaluated using 15 different applications using Integer linear programming and greedy programming techniques. The result shows that proposed approach reduced on average 84% in the case of IP and 81% in the case of Greedy programming.

In 2018, M. Linares-Vasquez et al [39] presents a GUI based energy optimization technique which produces color solutions optimizing energy consumption and contrast while using consistent colors with respect to the original color palette. Empirical evaluation shows that some of the solutions generated by GEMMA are able to achieve a good energy reduction while being acceptable by end-users.

2.1.2.2 Energy Consumption

Application's code level energy estimation-based research papers have been included to gather information about the energy estimation and consumption of various source code elements.

In 2013, proposed approach [40] focused on calculating the energy consumption at the source code level. This approach calculates the energy consumption by using combination of hardware-based power measurement and statistical modeling techniques. In 2014, [36] evaluated its proposed approach presented in [40] on 405 real world applications showing that on average applications spend 61% of their energy in idle states, network is the most energy consuming component, and only a few APIs dominate non-idle energy consumption.

In 2017, M. Wan et al [41] present technique to detect display related energy hotspots. Energy hotspots are the user interfaces of smartphone applications whose energy consumption is more than optimal. Evaluation shows that the presented technique can predict display energy consumption to within 14% of the ground truth and accurately rank display energy hotspots. Another approach presented in year 2019 [42] presented method level energy consumption and extracts the relevant energy consumption and execution traces. In 2020, M. Couto et al [43] studied application-level refactoring approach. The proposed work provides several findings that can guide developers in improving the energy efficiency of their code.

2.2 Analysis and Comparison

The primary criteria for selecting the test papers are test case prioritization. As the research area is inclined towards the testing of Android applications therefore the papers considering

Android applications for experimentation are considered. Apart from that the studies performing experimentation on java-based programs as considered as well.

Table 2.1 shows the comparisons of the code-based coverage criteria for test suite prioritization in the domain of Android applications.

After considering the code-based prioritization approaches, the papers relevant to the energy-based coverage has been included. As there is no such paper that focuses on energy aware TCP therefore, the other optimization related papers have been included as well. Table 2.2 shows the comparisons of the energy optimization techniques mentioned in literature. while Application's code level energy estimation-based research papers have been included to information about the energy estimation and consumption of various source code elements which are not directly a part of literature review.

TABLE 2.1: Literature Review: Code Based TCP

Ref No	Coverage Criteria	Evaluation Parameters	Weakness	Result
[8]	Statement	APFD	Presented evaluation on 2 GUI applications for prioritization but does not considers code level energy bugs	Multi-objective strategy performs better than two single strategies
[27]	Statement & Method	APFD	More complex methods are more fault prone, which may be difficult to detect. Coverage both elements still not ensure coverage of code level energy bugs	Techniques derived using basic and extended models can be more effective than total techniques & competitive with additional techniques
[28]	Functional Statement Branch Fault & Path	APF _{cd} C APF _{fn} C APF _{pt} C APF _{br} C APF _{fl} C	Focused on code level faults but does not considers possible energy bugs at code level	Enhanced method not only prioritize test cases but also recorded higher percentage of coverage for function & code coverage

Ref No	Coverage Criteria	Evaluation Parameters	Weakness	Result
[29]	Method & Statement	APFD	Aims to detect fault-prone source code only. Energy based faults are not considered	Evaluation shows that QTEP could improve existing coverage-based TCP techniques for both regression test cases and all test cases
[30]	Method & LOC	N/A	Uses total strategy and does not deals with energy based prioritization	No comparison done so no results are reported
[31]	Method & Branch Statement	APFD & M-spreading	The technique cannot be applied to the programs having no previous records. Focus only on source code based faults	Results show that additional strategies perform better in terms of APFD value while techniques that follow total strategies performed better considering M-spreading values
[32]	Path complexity & Branch	APFD	Code based energy leakage is not analyzed	The results show that proposed approach outperforms existing branch coverage-based approach in terms of APFD (Average Percentage of Faults Detected) up to 18% on average
[33]	Functional Statement & Branch Fault & Path	APF _{st} C APF _{br} C APF _{pa} C APF _{fn} C APFD	Does not covers possible bugs w.r.t energy leakage at any of the selected granularity level	Enhanced weighted method performs better than weighted method. But both show same result in AP _{fn} C value

Ref No	Coverage Criteria	Evaluation Parameters	Weakness	Result
[34]	Statement Branch & Method	APFD & APFDC	Energy leakage patterns are not uncovered by the presented approach	CCCP gives better results as compared to others. CCCP prioritization costs are found to be comparable to the additional test prioritization technique
[35]	Statement	APFD	Presented statement level defect prediction model, which is unable to uncover the possible statement level energy bugs	In the test case prioritization methods, the APFD performance of the additional strategy is preferable to max strategy and total strategy

TABLE 2.2: Literature Review: Energy Optimization

Ref No	Coverage Criteria	Evaluation Parameters	Weakness	Result
[37]	Energy consumption	Minimization time Percentage reduction	No explicit identification of energy bugs which they are handling. ILP based approach for minimizing test suite.	Evaluation result shows that the approach produces 95% energy efficient test suite in less than a second
[36]	Energy Consumption	Reduction percentage	No explicit identification of energy bugs which they are handling. ILP based approach for minimizing test suite	Results show that while maintaining high code coverage energy consumption has been reduced to on average of 5% to 48 %

Ref No	Coverage Criteria	Evaluation Parameters	Weakness	Result
[38]	Energy-aware coverage	Reduction Percentage Mutation analysis	Method level coverage. All categories of energy bugs are not considered. Executes the test case to calculate coverage which increase the cost. ILP based approach is not valid for large applications.	Result shows that the IP shows better results than greedy programming while maintaining test suite quality to reveal the great majority of energy bugs
[39]	GUI coverage (screen)	Battery percentage improvement	Focus only on GUI (coloring) based energy consumption. No code-based coverage criterion is used. Does not covers energy bugs/hotspots.	Empirical evaluation showed that some of the solutions generated by GEMMA are able to achieve a good energy reduction while being acceptable by end-users

From the above analysis, it has been shown that the energy-based test case prioritization is the least discussed domain. Most of the studies till now focuses on the energy optimization or energy estimation techniques for the Android application. Apart from that, code level energy bugs are not considered while performing energy optimization or calculating the energy estimation.

Chapter 3

Proposed Approach

In the literature survey, it has been identified that energy consumption is the least discussed topic in the domain of regression testing. After studying various TCP techniques, it has been concluded that these techniques focus on either white box or black box coverage based prioritization. While energy-inefficiency related studies focused on energy-based optimization or consumption at the coarse-grained level. Most of the energy aware test suite optimization techniques either execute the test cases to calculate coverage which increases the cost or uses the ILP approach which is not suitable for large applications.

To overcome the gap identified in the existing techniques, we proposed an energy aware TCP technique which identifies bug-prone paths and considers energy bug coverage as the major criterion for prioritizing the test cases. The path containing energy bug will be considered as the bug-prone path.

Considering these bug-prone paths, a new algorithm has been developed which uses the energy bug weight and statement coverage criteria to prioritize the test suite. Test paths, statement and bug coverage information are passed as input to the algorithm. The proposed algorithm uses both statement and bug coverage criteria simultaneously to compute the weight of the test paths. Then the algorithm generates prioritized list of test paths as the output.

3.1 Proposed Solution

We have proposed TCP technique that uses energy bug and statement coverage criteria for prioritizing the test suite. Figure 3.1 shows the proposed methodology for energy aware test suite prioritization. Test path's weight calculation process is considered as a sub-process of the prioritization algorithm. The prioritization algorithm uses the test suite, code coverage and energy bug coverage as input and returns the prioritized list of test cases. The output is then used in the evaluation process which takes fault coverage information as input.

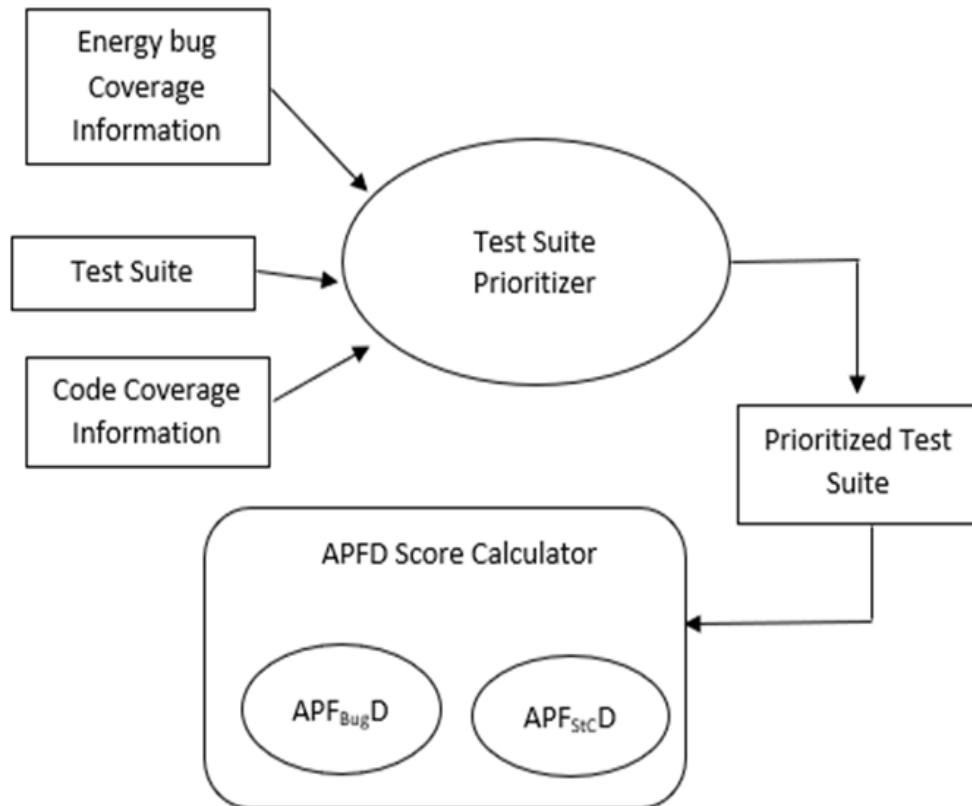


FIGURE 3.1: Context Diagram of the Proposed Approach

The prioritization algorithm takes test suite and coverage information and calculates test paths weights. The weight of entire test path is calculated using the following steps:

- Test path's weight calculation process returns the energy bug weight of test path (which is explained further in section 3.1.1.1)

- Energy bug weights are then normalized in the range of 0 to 1 using the min max normalization [44].
- Parallely statement coverage information of each test path is normalized in the similar range (i.e., 0 to 1).
- The whole test path's weight is calculated using the equation (3.1).

The test path having the maximum weight will be selected. As proposed algorithm follows the additional strategy, therefore, the residual coverage is calculated and then the process continues till all the test cases are prioritized. While, figure 3.2 depicts the complete flow of the prioritization algorithm.

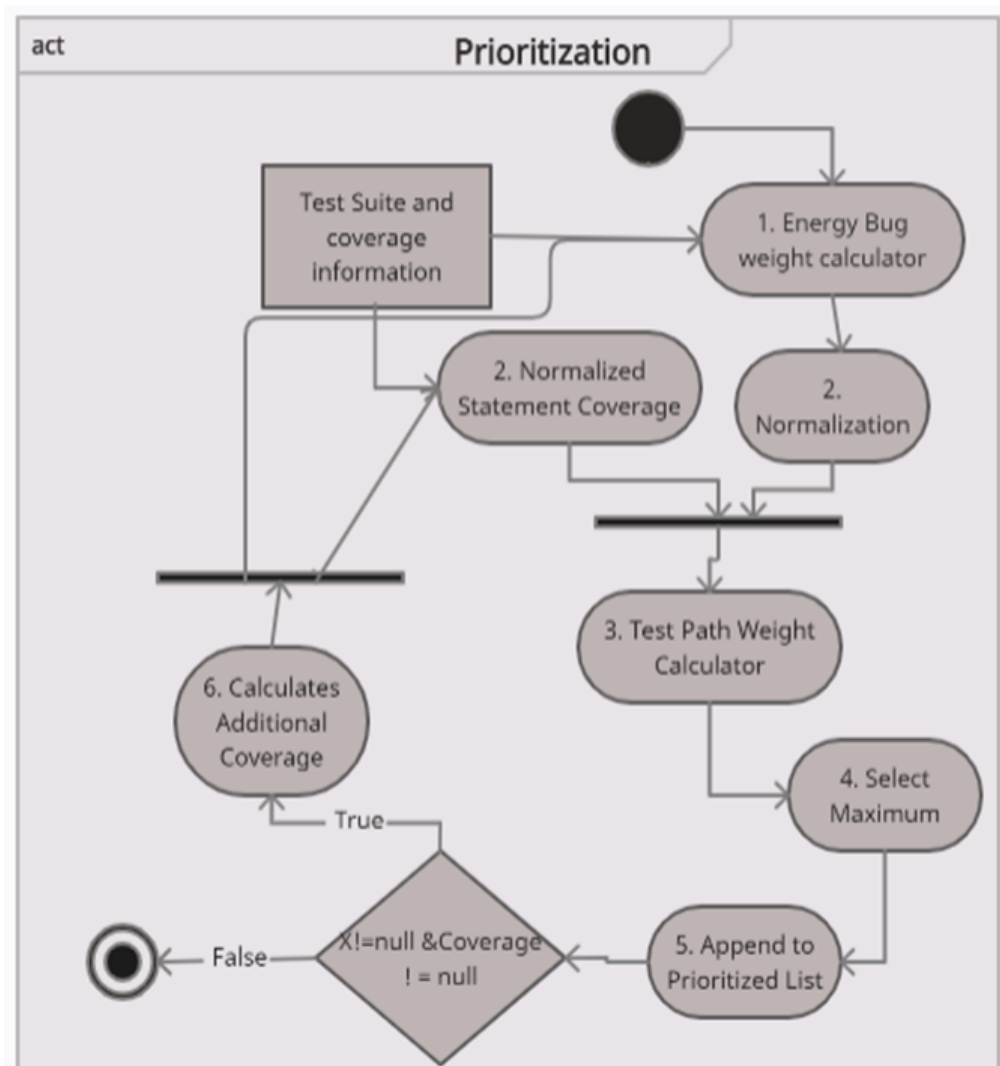


FIGURE 3.2: Flowchart of Prioritization Algorithm

The proposed approach can be applied during the maintenance testing phase of software development life cycle. It consists of the following three steps:

3.1.1 Prioritization Algorithm

The prioritization algorithm considers the energy bug weight and the statement coverage for prioritizing the test paths. The algorithm takes set of test paths, their coverage (energy bug and statement coverage) and resource information as input and generates the prioritized list of test paths. Energy-bug weight calculation is considered as the sub-part of the prioritization algorithm, using which the test path's weight is calculated.

3.1.1.1 Test Path's Weight Calculation

Weight Calculation phase takes the coverage information and the test paths as input and then calculates the respective weight of each test path. Test paths have been generated using Control Flow Graph (CFG). The source code of Android application has been taken as input using which the CFG is generated.

The CFG has been generated using the Auto-Tester tool [45]. It is a Java based tool used to generate the CFG from the source code. Once the CFG is generated, test paths are selected using the statement coverage criterion.

For elaborating the proposed methodology an example has been selected. Using the available source code, AutoTester tool generates the CFG as shown in the figure 3.3. The tool also generates test paths based on node coverage criteria where node represents the source code statements.

From the CFG, test paths have been created using one of the structural graph coverage criteria that is node coverage criteria. Auto tester tool also provides an option of generating test paths using different structural coverage criteria. Hence, the test paths are created using the auto tester tool by selecting the node coverage criteria is presented in table 3.1.

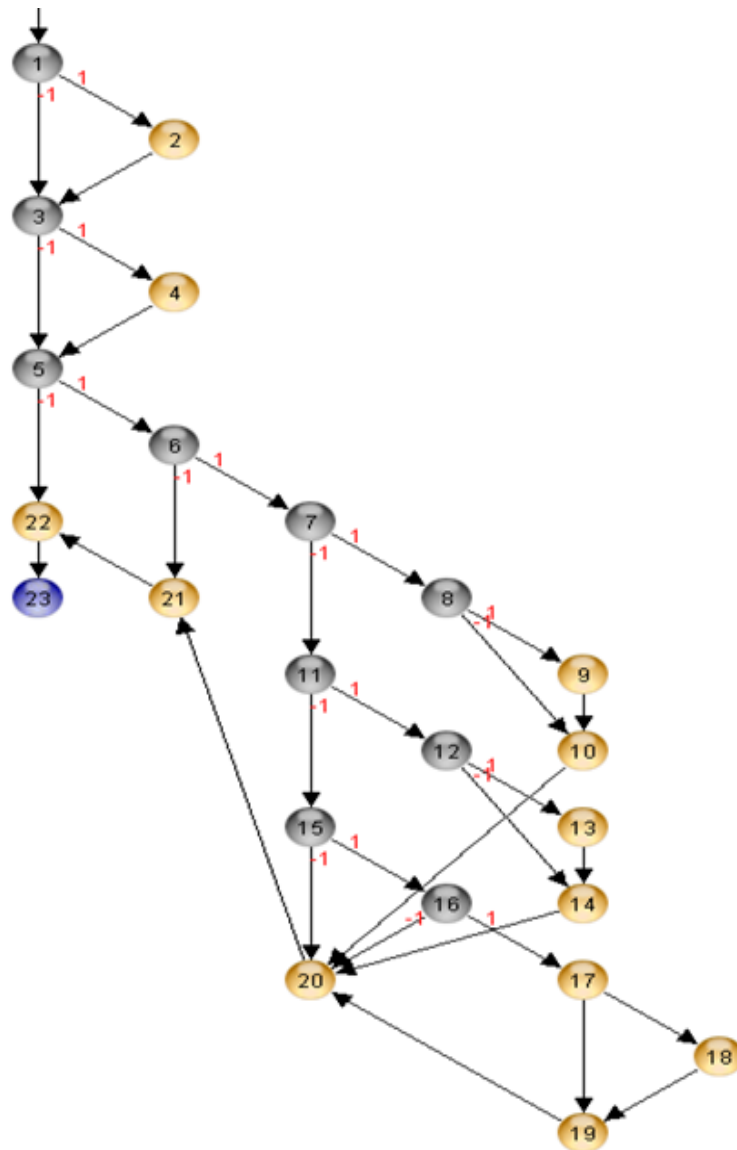


FIGURE 3.3: Example CFG

TABLE 3.1: Test Paths from Example CFG

Test Path No	Test Paths
TP1	1, 3, 5, 6, 7, 11, 15, 16, 17, 18, 19, 20, 21, 22
TP2	1, 3, 5, 6, 7, 11, 12, 13, 14, 19, 20, 21, 22
TP3	1, 3, 5, 6, 7, 8, 9, 10, 19, 20, 21, 22
TP4	1, 3, 5, 21, 22
TP5	1, 2, 3, 5, 21, 22

After creating CFG and the test paths, the nodes of CFG are then annotated with the acquisition, usage or release of the resources. The nodes of the CFG are annotated by statically analyzing the code. In the given CFG, two types of resources have been used that are wake lock and GPS. The Nodes in the example CFG is annotated and defined as follows:

- Wake lock has been acquired at Node 9 and 13
- Wake lock has been used at Node 8 and 12
- GPS has been acquired at Node 17

Bugs are identified and then the weight of each test path has been calculated using the following formula.

For a given set of test paths $tp_i = tp_1, tp_2, tp_3, \dots, tp_n$, weight of each path will be calculated based on bug type (as mentioned in section 3)

$$TP\text{-BugWeight} [tp_i] = \sum_{r=1}^n \{ \sum_{j=1}^3 [\text{cover}(tp_i, b_j) * bw_j] * \phi_r \} \quad (3.1)$$

Where;

n is the number of resources a test path covers.

i represents the number of test paths, j depicts the energy bug occurrence scenario and r is the number of resources a test path have.

tp_i are the i th test path among the test suite and b_j is the bug type tp_i covers.

$\text{cover}(tp_i, b_j)$ is 1 if it covers energy bugs b_j otherwise 0. b_j is the weight dedicated to bugs w.r.t to its occurrence type.

bw_j is the weight dedicated to bugs w.r.t to its occurrence type.

Here ϕ_r represents the weight assigned to the resource being used w.r.t to its energy consumption.

Scalar weight value is assigned to each type of bug as represented in Table 3.2. As A-Bug is consider as the severe bug (as resource is only acquired) therefore highest weight is assigned to it (i.e. 3). AU-Bug is less severe as compared to the A-Bug therefor weight is 2 for this type of bug. while AR-Bug is the bug with least severity so 1 weight has been assigned to it.

TABLE 3.2: Representing assigned weight w.r.t energy bug occurrence

Types	Bug occurrence scenario	Weight (bw)
A-Bug	Where any resource is acquired only but not used or released	3
AR-Bug	Where any resource is acquired and released but not used	1
AU-Bug	Where any resource is acquired and used but not released	2

These resource weights are calculated using the data collected on the basis of power consumption (in mW) of the API being used in the source code [46]. Table 3.3 shows the values assigned to the resource, showing that Wakelock consumes the maximum power.

TABLE 3.3: Resource Weights W.r.t Power Consumption

Sr. No	Resource Type	Energy Consumption	Weight ϕ
i	Bluetooth	211 mW	1
ii	GPS	408 mW	$1.93 \approx 2$

Sr. No	Resource Type	Energy Consumption	Weight ϕ
iii	Wi-Fi	483 mW	2.3
iv	Wake Lock	Between 500 to 2000mW	3

A test path, whose energy bug weight needs to be calculated, is passed to the weight calculation function. Using the formula mentioned in equation (3.1), the function computes the weight of each test path.

Algorithm 1 represents the algorithm used to calculate the energy bug weight for each test path.

Algorithm 1 Energy Bug Weight Calculation(t)

Input:

- Test Path t
- $|r|$: Integer value representing how many types of resource bugs a test path contains
- b - cov: Coverage vector such that for each test path $t \in T$, b - cov(t) the set of energy bugs covered by executing P against t

Output:

Weight of the respective test path t

Declare:

TPWeight: weight of test path

TP1[]: temporary array for storing weights

```

1: begin
2: if  $|r| > 1$  then
3:   for  $j \leftarrow 1:n$  do
4:     for  $i \leftarrow 1:3$  do
5:        $TP1[t] \leftarrow (\text{cover}(t, b_i) * bw_i) * \phi_j$ 
6:      $TPWeight \leftarrow TPWeight + TP1[t]$ 
7: else
8:   for  $i \leftarrow 1:3$  do
9:      $TPWeight \leftarrow (\text{cover}(t, b_i) * bw_i) * \phi_i$ 
10: return TPWeight
11: end

```

The algorithm checks for the number of resources a test path has. If the function covers only one resource than the algorithm checks for its bug type w.r.t to that resource and the energy bug weight is calculated. In case if the test path covers multiple resources than the energy bug weight w.r.t to each resource type is calculated and then summed up to form the weight of entire test path.

Continuing to the given example the energy bugs have been identified using which the weight w.r.t to bugs has been calculated. Table 3.4 shows the energy bug weight calculated for each test paths.

TABLE 3.4: Energy-Bug Coverage Details:Example

Test Path No	Statement Coverage	Identified Bug	No of Bugs	Resource Used	Bug Weight
TP1	1, 3, 5, 6, 7, 11, 15, 16, 17, 18, 19, 20, 21, 22	AcqBug	1	GPS	6
TP2	1, 3, 5, 6, 7, 11, 12, 13, 14, 19, 20, 21, 22	AcqBug	1	Wake Lock	12
TP3	1, 3, 5, 6, 7, 8, 9, 10, 11, 19, 20, 21, 22	AcqBug	1	Wake Lock	12
TP4	1, 3, 4, 5, 21, 22	No Bug	0	–	0
TP5	1, 2, 3, 5, 21, 22	No Bug	0	–	0

The complete weight of a test path is calculated by summing the normalized energy bug weight and normalized statement coverage as follows:

$$\text{TPWeight} [tp_i] = (\alpha) \text{NBW} [tp_i] + (1 - \alpha) \text{NSC} [tp_i]$$

(3.2)

In equation (3.2) , α is a scalar variable which is used to give importance to a specific value. In the proposed algorithm, initially α have the value of 0.5 which depicts that equal importance will be given to the energy bug weight and statement coverage. While these values can be altered in order to give more importance to any of the selected criteria.

The proposed algorithm uses the additional strategy to compute the weight of test path. Additional strategy states that the energy bug or statements covered by a test case should not be used in the weight calculations of other test paths. Test path with the highest weight will be selected. After selecting a test path, the weight of the remaining test paths will be calculated again and hence the process continues. The proposed algorithm following the additional strategy is elaborated in the algorithm 2.

Algorithm 2 Prioritization Algorithm

Input:

T: Set of test paths of program P

StatementCoverage: Set of statements in P covered by all test paths in T

BugCoverage: Set of energy bugs in P covered by all test paths in T'

s-cov: Coverage vector such that for each test path $t \in T$, s-cov(t) is the set of statements covered by executing P against tb-cov: Coverage vector such that for each test path $t \in T$, b-cov (t) the set of energy bugs covered by executing P against t**Output:**

PrT: array of sequenced test path such that

- Each test PrT belongs to T
- Each test in T appears exactly once in PrT

Declare:

TPWeight: weight of test path

TP1[]: temporary array for storing weights

```

1: begin
2:  $X' \leftarrow T$ 
3: for  $t_i \in X' \leftarrow 1:n$  do
4:    $W [t_i] \leftarrow \text{BugWeight}(t_i)$ 
5:    $NW [t_i] \leftarrow \text{Normalization} (W (t_i))$ 
6:    $NSC [t_i] \leftarrow \text{Normalization} (| \text{s-cov}(t_i) |)$ 
7:    $TPW [t_i] \leftarrow (\alpha) NW [t_i] + (1- \alpha) NSC [t_i]$ 

```

```

8: while  $X' \neq \phi$ , BugCoverage  $\neq \phi$  and StatementCoverage  $\neq \phi$  do
9:   Find  $t \in X'$  Such that  $|TPW[t]| \geq |TPW[u]|$  for all  $u \in X'$ ,  $u \neq t$ 
10:  Set PrT.append (t) and  $X' \leftarrow X' \setminus \{t\}$ 
11:  StatementCoverage  $\leftarrow$  StatementCoverage  $\setminus |s\text{-cov}(t)|$ 
12:  BugCoverage  $\leftarrow$  BugCoverage  $\setminus |b\text{-cov}(t)|$ 
13:  for  $t_i \in X' \leftarrow 1:n$  do
14:    for  $t_i \in \text{PrT} \leftarrow 1:n$  do
15:      if  $b\text{-cov}(t_i) \neq \text{BugCoverage}$  then
16:         $b\text{-cov}(t_i) \leftarrow 0$ 
17:      if  $s\text{-cov}(t_i) \neq \text{StatementCoverage}$  then
18:         $s\text{-cov}(t_i) \leftarrow 0$ 
19:       $W[t_i] \leftarrow \text{BugWeight}(t_i)$ 
20:       $NW[t_i] \leftarrow \text{Normalization}(W(t_i))$ 
21:       $NSC[t_i] \leftarrow \text{Normalization}(|s\text{-cov}(t_i)|)$ 
22:       $TPW[t_i] \leftarrow (\alpha)NW[t_i] + (1-\alpha)NSC[t_i]$ 
23: Return PrT
24: end

```

The weight of the test paths (for the selected example) is expressed in table 3.5. The table represents bug weight calculated in table 3.4 and respective statement coverage. Total test path weight is then calculated based on the normalized weights of energy bugs and statement coverage.

TABLE 3.5: Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-Step I:Example

Test Path No	Bug Weight	Normalized Bug Weight	Statement Coverage	Normalized Statement Coverage	Total Weight
TP1	6	0.5	14	1	0.75
TP2	12	1	13	0.875	0.93
TP3	12	1	12	0.75	0.87
TP4	0	0	6	0	0
TP5	0	0	6	0	0

Here the weight of the test path <TP2> is maximum therefore TP2 is selected as the first test in the prioritization order. When any test path is selected, the algorithm updates the coverage information of the test paths following the additional

strategy and again calculates the individual weight of test paths.

After selecting a test path, the coverage information is re-calculated. Using the additional coverage information the test path weight is calculated again as shown in table 3.6.

TABLE 3.6: Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-Step II:Example

Test Path No	Bug Weight	Normalized Bug Weight	Additional Statement Coverage	Normalized Statement Coverage	Total Weight
TP1	6	1	4	1	1
TP3	0	0	3	0.66	0.33
TP4	0	0	1	0	0
TP5	0	0	1	0	0

<TP1> is selected, as it has the highest weight value among all and is placed in the prioritization list after TP1. As the coverage information and test paths are not null therefore recalculate the test path weights based on the additional coverage information.

TABLE 3.7: Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-Step III:Example

Test Path No	Bug Weight	Normalized Bug Weight	Additional Statement Coverage	Normalized Statement Coverage	Total Weight
TP3	0	0	3	1	0.5
TP4	0	0	1	0	0
TP5	0	0	1	0	0

<TP3> is selected as it has the highest value as shown in 3.7. The whole process continues until all the test paths are prioritized and all the energy bugs and statements are covered.

Hence the prioritized list has been generated which is { TP2, TP1, TP3, TP4, TP5 }.

3.1.2 Evaluation

Average Percentage of Fault Detection (APFD) is used to evaluate the effectiveness of prioritization techniques. APFD is the measure of how early faults are detected in the testing process by the test suite [5]. Following formula is used to calculate the APFD:

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + \dots + TF_n}{nm} + \frac{1}{2n} \quad (3.3)$$

Where T is the test suite containing n test cases and F is the set of m faults revealed by T. For ordering T', TF_i is the order of the first test case that reveals the i th fault F_i . Further details are mentioned in chapter 5 (section 5.3).

Chapter 4

Implementation

This chapter discusses the implementation details of proposed algorithm. The proposed algorithm has been implemented using the Eclipse software in Java language.

4.1 Implementation Details

The class diagram of proposed algorithm is shown in the figure 4.1. The code includes three classes. Classes are designed to initialize the set of test paths and coverage information, calculating the energy bug weights and then create a list of prioritized test paths following the proposed algorithm.

The classes used in the class diagram and their relations are elaborated as follows.

- **DataRetrieval class:** This class initializes the set of test paths and coverage information which is then used in the prioritization class. This class also has normalization function which is used to normalize the data values passed to it.
- **WeightClass:** This class calculates the energy bug weight of each test path based on the resource and the related energy bugs it contain.

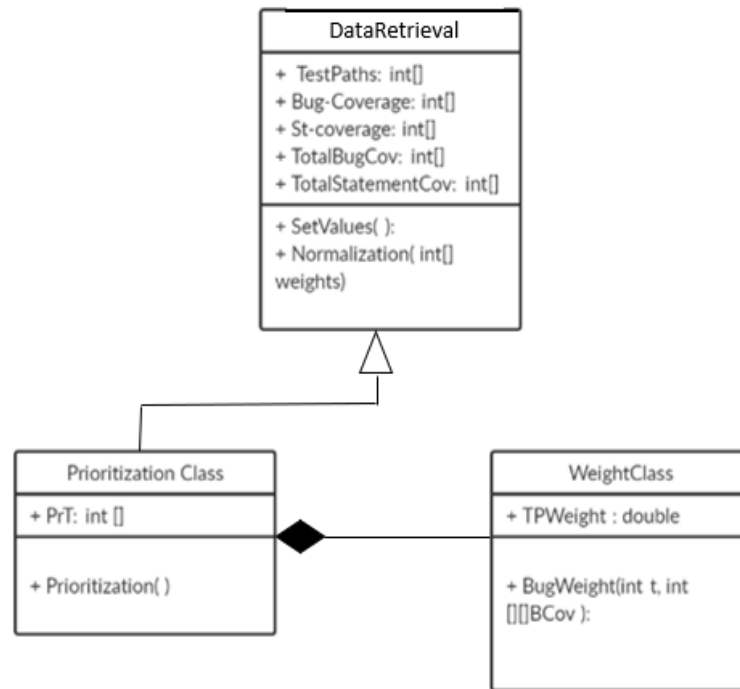


FIGURE 4.1: Class Diagram of Proposed Algorithm

- **PrioritizationClass:** This class implements the proposed prioritization algorithm on the set of test paths and generates the prioritized list.

The prioritization class uses the values initialized in the GetValues class, therefore, this class inherits the GetValues class in order to access the data members and member functions. While the prioritization algorithm involves calculating the energy bug weights of every test path in every iteration therefore the WeightClass is considered as a sub-part of the prioritization class. Hence composition relationship exists among both the classes.

The methods involved in the above defined classes are discussed below:

- **SetValues ():** This function initializes the data members of the class on user defined inputs which is further used in the prioritization method.
- **Normalization ():** Weight of the test path is calculated using the energy bug weight and the statement coverage information. Both of these values are normalized on the scale of 0 to 1 using the min-max normalization technique. This function is used to generate the normalized values.

- **BugWeight ()**: This function takes test path and its related energy bug coverage information and returns the weight of the respective test path. The function calculates the energy bug weights of the test path using the equation (3.1) (mentioned in chapter 3).
- **Prioritization ()**: This method performs the proposed prioritization algorithm which makes use of the other methods for generating the prioritized test cases.

Initially, the data members inside the `GetValues` are initialized using the `SetValue ()`. After initializing the values, `Prioritization ()` starts execution using the initialized values. `BugWeight ()` returns the energy bug weight of each test path which is then normalized using the `Normalization ()`. Once the test path has been selected, the process re-computes the energy bug weights using the residual values. The process of normalization occurs again in order to compute weight of remaining test paths. This process continues until the whole test suite is prioritized and every energy bug and statement is covered.

4.2 Usage Details

The proposed algorithm takes test paths, the total coverage information and the energy bug and statement coverage of the test paths from the file in `DataRetrieval` class.

Figure 4.2 shows the input file format for the example program, in which the coverage information is input in comma (,) separated way. While the test paths energy bug and statement coverage is added in separate lines.

Figure 4.3 shows the input format for statement coverage of test paths where each line shows the coverage of a test path. whereas in a line, statements are separated by commas. In the code this file data is extracted and stored in a two-dimensional array.

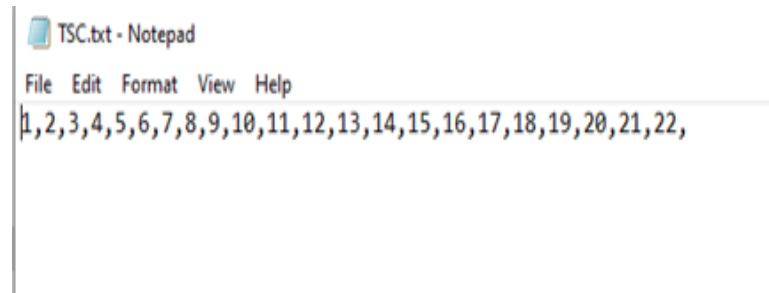


FIGURE 4.2: File Format: Input Screen-I

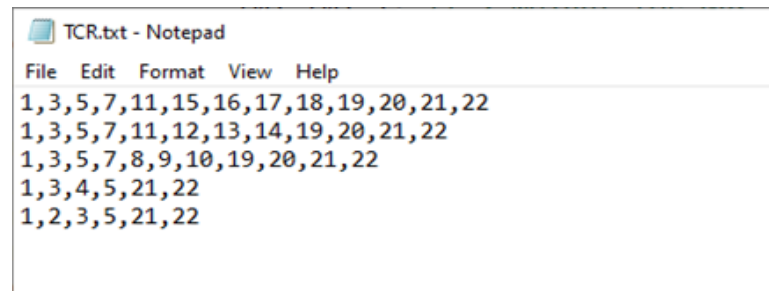


FIGURE 4.3: File Format: Input Screen-II

Considering the test paths coverage information, the algorithm calculates the energy bug weight. For computing the final weight of the test paths, the proposed algorithm uses energy bug weight and statement coverage of that test path. Among all the test paths, the path with the maximum weight is selected. While calculating the weight of entire test path, the proposed algorithm gives equal weight-age to the energy bug coverage and statement coverage.

Once the test path is selected, the algorithm updates the total coverage information and then re-calculates the weights of test paths. This process continues until all the test paths are prioritized. The algorithm generates the prioritized list of the test paths on the console shown in the figure 4.4.

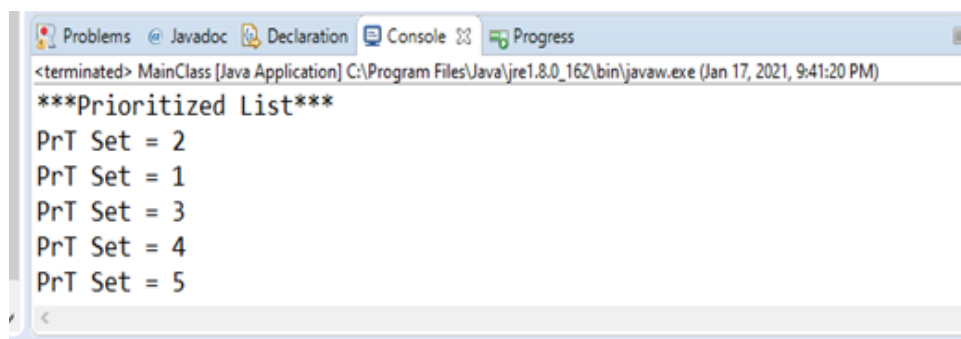


FIGURE 4.4: Console Output Screen

4.3 Mapping Example

For elaborating the implementation details, we have chosen hypothetical example whose input and output is explained in steps below.

4.3.1 Input Details

Initially, coverage information is input using the text files (format as mentioned in section 4.2). The coverage information includes number of test paths, total statement coverage, total bug coverage, test path's statement and bug coverage.

Figure 4.5 shows the file format for total statement coverage an algorithm needs to check. The file should have one row and all the statements should be written in comma (,) separated format. As the algorithm follows the node coverage criterion therefore, the statements are considered as the nodes of CFG.

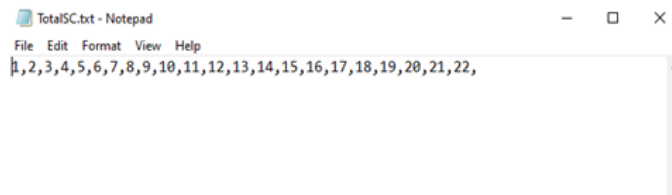


FIGURE 4.5: File Format: Total Statement Coverage

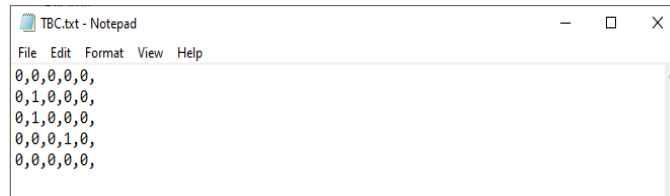
The total energy bug coverage information 's file format is shown in figure 4.6. The file shows the data regarding the five different resource types starting from Bluetooth till WakeLock (see table 3.2). Data should be added in a single row but have four different values representing the total bugs w.r.t each resource type.



FIGURE 4.6: File Format: Total Energy Bug Coverage

Now the next step is to input the coverage information for each test path. Energy bug coverage and statement coverage data of a test path are inputted using two

separate files. Figure 4.7 and 4.8 represents the test paths energy bug coverage and statement coverage respectively.

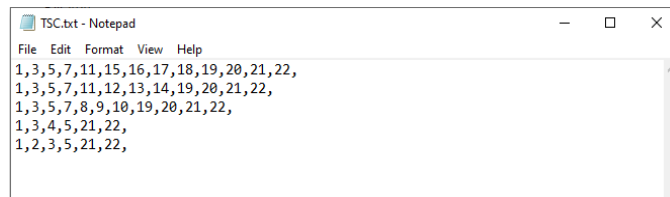


```

TBC.txt - Notepad
File Edit Format View Help
0,0,0,0,0,
0,1,0,0,0,
0,1,0,0,0,
0,0,0,1,0,
0,0,0,0,0,

```

FIGURE 4.7: File Format: Test Path Energy Bug Coverage



```

TSC.txt - Notepad
File Edit Format View Help
1,3,5,7,11,15,16,17,18,19,20,21,22,
1,3,5,7,11,12,13,14,19,20,21,22,
1,3,5,7,8,9,10,19,20,21,22,
1,3,4,5,21,22,
1,2,3,5,21,22,

```

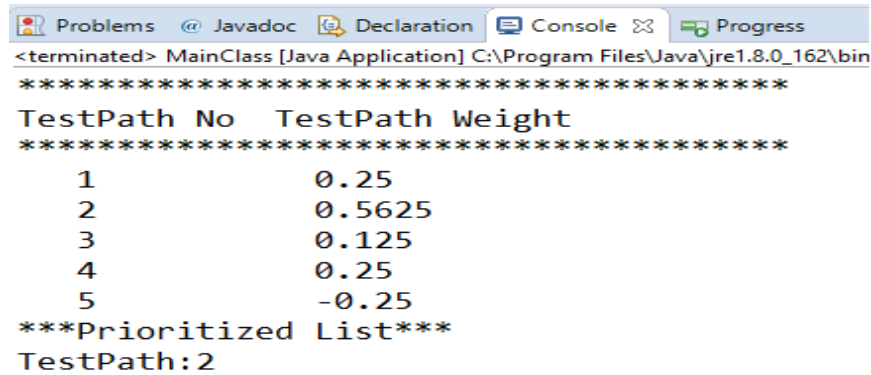
FIGURE 4.8: File Format: Test Path Statement Coverage

Each row represents the test path and data separated by commas represents the coverage information of that test path. Here in figure 4.7, test path 1 and 5 covers no energy bug, test path 2 covers a GPS bug of type A-Bug, test path 3 covers the same bug as that of test path 2, test path 4 covers wake lock bug of type A-Bug. Here the number represents the bug types as discussed in table 1.2 (chapter 1).

4.3.2 Output Details

According to the prioritization algorithm (defined in chapter 3), energy bug weight for each test path is calculated and normalized in the range of 0-1. After-wards using the normalized statement coverage, the total weight of a test path is calculated. Then the test path with the highest weight has been assigned the highest priority among all.

Figure 4.9 shows the initial test path's total weight and the selection of test path among the five test paths. These weights are calculated based on the algorithm explained in the chapter 3. Using the statement and the energy bug coverage weights, the weight of the test path is calculated which is further used for prioritization.



```

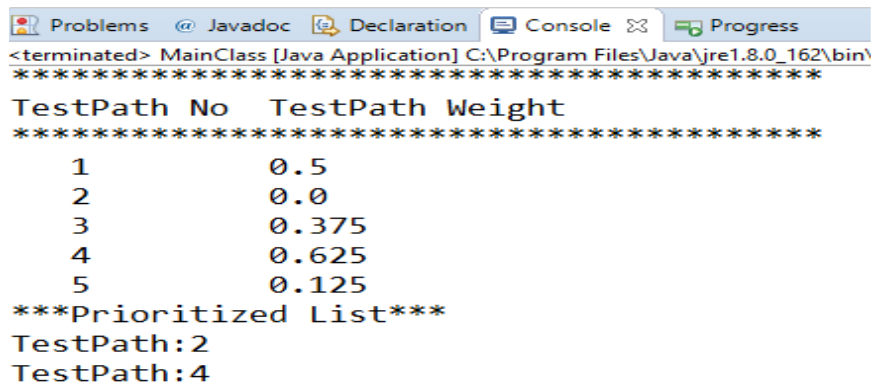
Problems @ Javadoc Declaration Console Progress
<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_162\bin
*****
TestPath No  TestPath Weight
*****
    1          0.25
    2          0.5625
    3          0.125
    4          0.25
    5         -0.25
***Prioritized List***
TestPath:2

```

FIGURE 4.9: Output: Prioritized List-I

In the above figure the test path 2 has the highest weight therefore, TP2 is selected and added to the list. As the algorithm follows the additional prioritization strategy so, the residual coverage of each test path is calculated. Based on the residual coverage information the total weight of the remaining test paths are re-calculated. Test Path with the highest weight is then appended to the prioritization list. This process continues until each test path is added to the prioritization list and all the energy bugs and statements are covered.

Figure 4.10, shows the selection of next test path w.r.t to the total weight.



```

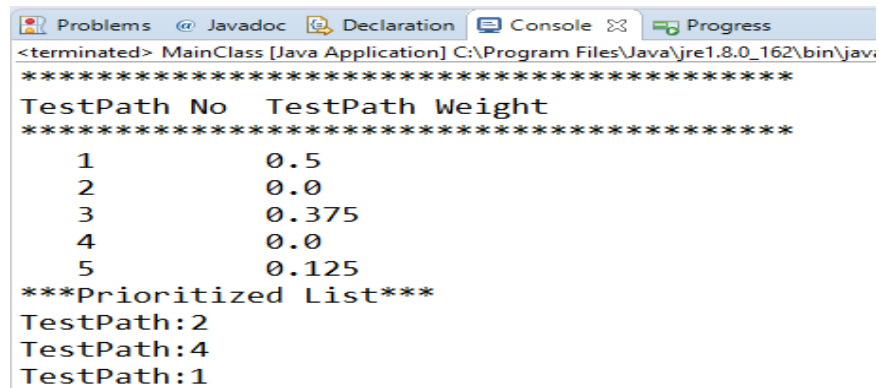
Problems @ Javadoc Declaration Console Progress
<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_162\bin
*****
TestPath No  TestPath Weight
*****
    1          0.5
    2          0.0
    3          0.375
    4          0.625
    5          0.125
***Prioritized List***
TestPath:2
TestPath:4

```

FIGURE 4.10: Output: Prioritized List-II

As the weight of TP4 is the highest so, it is appended to the priority list. As the algorithm follows the additional approach, therefore, next weights are calculated based on additional coverage information.

After selecting the test path 4, the selection of next test path (i.e. test path 1) is based on its weight which is shown in figure 4.11.



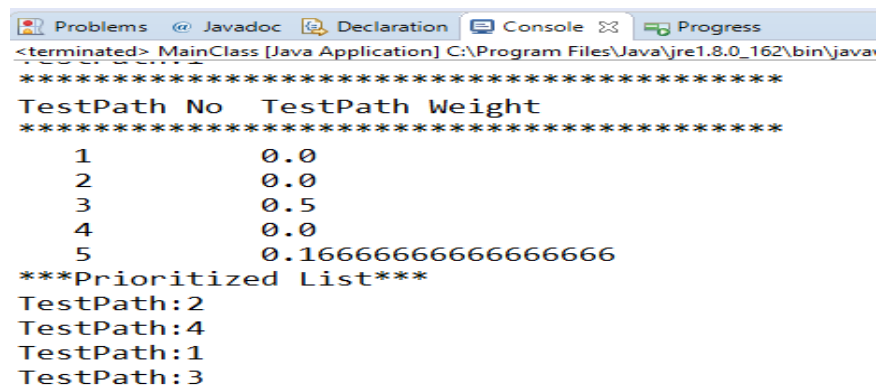
```

<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe
*****
TestPath No  TestPath Weight
*****
    1          0.5
    2          0.0
    3         0.375
    4          0.0
    5         0.125
***Prioritized List***
TestPath:2
TestPath:4
TestPath:1

```

FIGURE 4.11: Output: Prioritized List-III

Figure 4.12 and 4.13 shows the selection of next test paths of prioritized list.

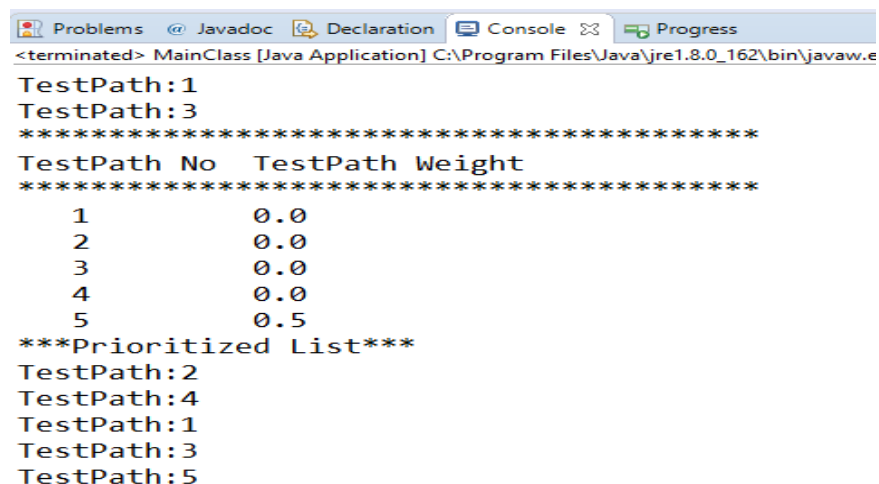


```

<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe
*****
TestPath No  TestPath Weight
*****
    1          0.0
    2          0.0
    3          0.5
    4          0.0
    5         0.16666666666666666
***Prioritized List***
TestPath:2
TestPath:4
TestPath:1
TestPath:3

```

FIGURE 4.12: Output: Prioritized List-IV



```

<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe
TestPath:1
TestPath:3
*****
TestPath No  TestPath Weight
*****
    1          0.0
    2          0.0
    3          0.0
    4          0.0
    5          0.5
***Prioritized List***
TestPath:2
TestPath:4
TestPath:1
TestPath:3
TestPath:5

```

FIGURE 4.13: Output: Prioritized List-V

At the end of figure 4.13, the prioritized test suite is displayed and the process terminates here.

Chapter 5

Results and Discussion

This chapter describes the results of the experiments done on different subject programs. For the purpose of evaluating the proposed prioritization algorithm, 10 different Android applications have been selected. These applications are selected from the studies exists in literature. The source code of these applications has been downloaded from [47].

The proposed prioritization technique works on the fine-grained level (i.e., Statement level). We have generated the Control Flow Graphs of the functions where each node in the graph represents the line of code inside the function. Test paths of the CFG are then generated by applying the node coverage criterion. The node coverage criterion is the graph coverage criteria which covers all the nodes of the graph. These test paths represent the execution sequence of the statements inside the function. We have selected those functions in which any resource has been acquired, used or released at statement level.

Whenever any resource is acquired in the function it is assumed that the resource will be properly released after using it. If the developer acquires a resource in its code but does not release it at the end, then energy bug arises. Any path containing the energy bug is considered as bug-prone path. From the available test paths, the bug-prone paths are identified and their energy bug weights have been calculated.

Test path's weights are calculated using the additional strategy based on energy bug weight and the statement coverage. Using these weights, the prioritized list of the test paths is generated.

5.1 Subject Programs

For comparing our proposed approach, we have to choose the paper which covers energy bugs in Android applications therefore, the most relevant paper 'energy aware test-suite minimization' is selected. Out of 15 applications discussed in the paper [38], only 6 applications are relevant to our proposed approach. As our approach consider only five resource types that are commonly used in the Android applications (explained in table 3.3) so, those applications which do not utilize these resources are considered as irrelevant to our approach. Other four applications are selected randomly from the literature.

For evaluating the proposed technique, we have used 10 different open-source Android applications, discussed as follows:

5.1.1 AndroidRun

AndroidRun is an assistance application for runners and bikers. It provides the functionality of calculating distance, instant speed and the average speed. This application is able to log all the data along with the location information.

The source code of this application can be downloaded from [48].

5.1.2 Jamendo

Jamendo is Android based media player application. This application provides its user to play any song, at any time, on any device, without any interruptions or limitations. This application uses Wi-Fi and Wake lock resources.

The source code of this application can be downloaded from [49].

5.1.3 OpenCamera

Open Camera is an Open-Source Camera app for Android smartphones. This is multi-functional camera-based application. This application provides additional functionality of GPS location tagging (geotagging) of photos and video.

The source code of this application can be downloaded from [50].

5.1.4 A2DPVolume

A2DPVolume is the automatic media volume adjuster application. This adjusts the volume on connection and resets it after disconnection. The application is primarily intended for the vehicles Bluetooth system. This application keeps track of the location using which exiting car mode can also be enabled when person leaves the car.

The source code of this application can be downloaded from [51].

5.1.5 Apollo

Apollo is the java-based library which is used for defining service routes and manages request/reply handlers. Apollo is used at Spotify when writing microservices. Apollo includes modules such as an HTTP server and a URI routing system, making it trivial to implement restful API services. This application uses wake lock.

The source code of this application can be downloaded from [52].

5.1.6 Andromatic

Andromatic is the java based Android application to automate the actions based on user-defined triggers. This application uses Android location service.

The source code of this application can be downloaded from [53].

5.1.7 K9Mail

K9Mail is an open-source email platform for Android users. This application support multiple email accounts based on POP3, IMAP and pushIMAP. K9Mail also have encryption mechanism embedded in order to provide secure data transfer.

The source code of this application can be downloaded from [54].

5.1.8 MyTracks

Mytracks is an open-source GPS based location tracking application. This application provides the option of inserting image-based photo markers.

The source code of this application can be downloaded from [55].

5.1.9 Find3

Find3 application allows to performs constant scans for Bluetooth and WiFi signals and levels that can be associated with certain locations in your home.

It is an open-source application that can be downloaded from [56].

5.1.10 WiFiGPSLocation

WiFiGPSLocation is an Android service to simplify duty-cycling of the GPS receiver when a user is not mobile. The WiFiGPSLocation application runs as an Android Service on the phone.

The source code of this application can be downloaded from [57].

5.1.11 Features of Subject Program

The table 5.1 describes the detailed features of the applications used for the experiment.

TABLE 5.1: TCP Resource, CFG Test Paths and Bug-prone paths in each Android applications

Applications	Lines of Code	Resource Used	CFG Paths	Test	Bug-Prone Paths
AndroinRun	1021	GPS	9		7
Jamendo	8709	Wi-Fi	5		4
Open Camera	15,064	GPS	6		6
A2DP volume	6,670	Wi-Fi, GPS, Bluetooth	25		23
Apollo	20,520	Wake lock	8		5
Andromatic	2,156	GPS	5		5
K9 Mail	71,816	Wake lock	4		4
MyTracks	35,039	GPS, GoogleMap	5		3
Find3	14,819	Wake lock, Wi-Fi, GPS	9		7
WiFiGPS Location	19,293	Wake lock, Wi-Fi, GPS	22		12

Figure 5.1 represents the resource usage in subject programs, showing that GPS is the mostly used resource showing that 7 out of 10 applications used GPS resource. Wi-Fi is the second most used resource in the selected applications. While Bluetooth and the GoogleMaps API is the least used resource.

5.2 Prioritization Example Mapping on Android-Run Application

AndroidRun is an assistance application for runners and bikers. It provides the functionality of calculating distance, instant speed and the average speed using GPS service. Two functions from this application is selected to illustrates the proposed algorithm.

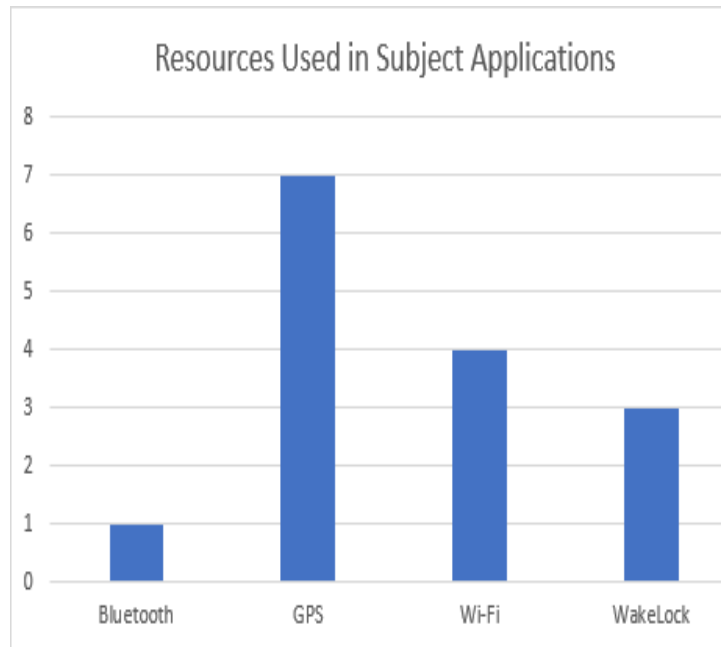


FIGURE 5.1: Resource Usage in Subject Programs

5.2.1 onCreate():AndroidRun Application

By statically analyzing the source code of the selected AndroidRun application, onCreate function is considered which uses GPS resource.

Firstly, the CFG of the selected Function has been created using AutoTester tool [45]. The figure 5.2 shows the CFG generated using the tool.

After generating the CFG, test path will be generated using the same tool by applying one of the graph coverage criteria which is node coverage criterion. Figure 5.3 shows the test paths generated from the CFG.

OnCreate function of AndroidRun is statically analyzed and the nodes are annotated as follows:

- GPS is acquired at node 2 and node 8
- GPS is used at node 6

Table 5.2 shows the coverage information of the selected method. The coverage details includes the bug type, resource used and the statement coverage.

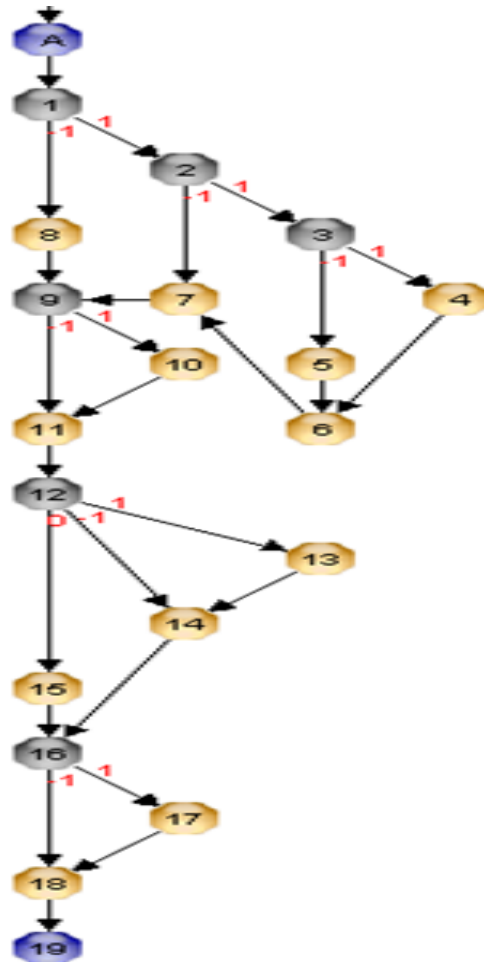


FIGURE 5.2: CFG of onCreate():AndroidRun Application

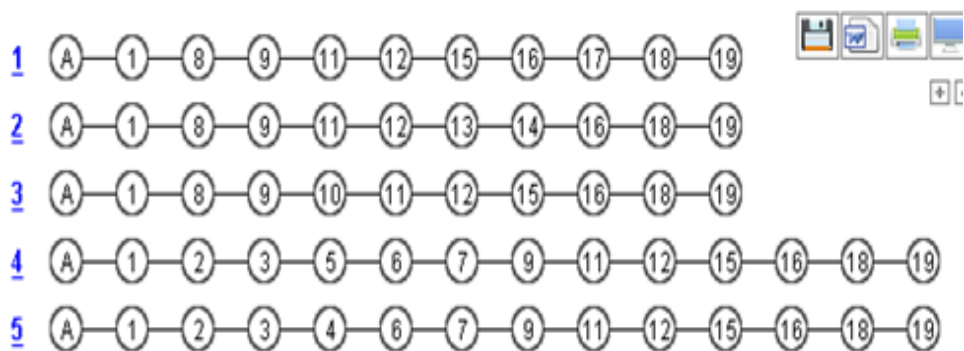


FIGURE 5.3: Test Paths of onCreate():AndroidRun Application

Initially, the energy bug weights of each test paths will be calculated using the available coverage information.

The test path’s weight will be computed by adding the normalized energy bug weight and statement coverage. The values are normalized to the range of 0 to 1

TABLE 5.2: TCP Resource, CFG Test Paths and Bug-prone paths in each Android applications

Test Paths	Energy Covered	Bug No	of Bugs	Resource Used	Statement Covered
TP1	AcqBug	1		GPS	1, 8, 9, 11, 12, 15, 16, 17, 18, 19
TP2	AcqBug	1		GPS	1, 8, 9, 11, 12, 13, 14, 16, 18, 19
TP3	AcqBug	1		GPS	1, 8, 9, 10, 11, 12, 15, 16, 18, 19
TP4	AU-Bug	1		GPS	1, 2, 3, 5, 6, 7, 9, 11, 12, 15, 16, 18, 19
TP5	AU-Bug	1		GPS	1, 2, 3, 4, 6, 7, 9, 11, 12, 15, 16, 18, 19

for both statement coverage and energy bug weights.

Table 5.3 shows the energy bug weights of test paths, its normalized values, normalized statement coverage and the weight of the entire test path.

TABLE 5.3: Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step I

Test Path No	Energy Bug Weight	Normalized Energy Bug Weight	Statement Coverage	Normalized Statement Coverage	Test Path Weight
TP1	6	1	10	0	0.5
TP2	6	1	10	0	0.5
TP3	6	1	10	0	0.5
TP4	4	0	13	1	0.5
TP5	4	0	13	1	0.5

As the test path weight of each test path is equal therefore, we can select any of them therefore, we have selected TP1.

So, $\langle \text{PrT} \rangle = \{ \text{TP1} \}$

As the prioritization algorithm follows the additional strategy, therefore for calculating the weights of test paths, additional coverage information will be required (mentioned in table 5.4)

Table 5.4 shows the additional coverage information of the selected method, after appending TP1 in prioritized test suite.

TABLE 5.4: Additional Coverage Information of Test Paths of onCreate(): AndroidRun Application

Test Paths	Additional Energy Covered	Bug	Additional Statement Covered
TP2	0		2
TP3	0		1
TP4	AU-Bug		5
TP5	AU-Bug		5

The process of test path's weight calculation continues until all the test paths will be prioritized. Table 5.5 shows the weights of test paths. TP4 and TP5 have the

TABLE 5.5: Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step II

Test Path No	Additional Energy Bug Weight	Normalized Energy Bug Weight	Additional Statement Coverage	Normalized Statement Coverage	Test Path Weight
TP2	0	0	2	0.25	0.125

Test Path No	Additional Energy Bug Weight	Normalized Energy Bug Weight	Additional Statement Coverage	Normalized Statement Coverage	Test Path Weight
TP3	0	0	1	0	0
TP4	4	1	5	1	1
TP5	4	1	5	1	1

highest weights so choose among any of them and append to the prioritized list.

So, $\langle \text{PrT} \rangle = \{ \text{TP1}, \text{TP4} \}$

After selecting the TP4, the residual coverage information is updated shown in table 5.6.

TABLE 5.6: Additional Coverage Information of Test Paths of onCreate(): AndroidRun Application -II

Test Paths	Additional Energy Bug Covered	Additional Statement Covered
TP2	0	2
TP3	0	1
TP5	0	1

All the Energy Bugs have been covered but statements are still not covered and all test paths are not prioritized. Therefore, repeat the process of weight calculation as shown in table 5.7.

From table 5.7, it can be seen that TP2 has the highest weight so TP2 is selected and appended to the list.

So, $\langle \text{PrT} \rangle = \{ \text{TP1}, \text{TP4}, \text{TP2} \}$

TABLE 5.7: Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step III

Test Path No	Additional Energy Bug Weight	Normalized Energy Bug Weight	Additional Statement Coverage	Normalized Statement Coverage	Test Path Weight
TP2	0	0	2	0.33	0.33
TP3	0	0	1	0	0
TP5	0	0	1	0	0

Table 5.8 shows the additional information after appending TP2 to the prioritization list.

TABLE 5.8: Additional Coverage Information of Test Paths of onCreate(): AndroidRun Application -III

Test Paths	Additional Energy Covered	Bug Covered	Additional Statement Covered
TP3	0		1
TP5	0		1

TABLE 5.9: Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step IV

Test Path No	Additional Energy Bug Weight	Normalized Energy Bug Weight	Additional Statement Coverage	Normalized Statement Coverage	Test Path Weight
TP3	0	0	1	1	0.5
TP5	0	0	1	1	0.5

Both the remaining test paths (in table 5.9) have the same weight and covers different statements therefore they are selected one by one.

Hence the prioritized list will be,

$$\langle \text{PrT} \rangle = \{ \text{TP1}, \text{TP4}, \text{TP2}, \text{TP3}, \text{TP5} \}$$

5.2.2 onLoctaionChange():AndroidRun Application

Another function from the same application uses the GPS resource as well. Therefore, the complete prioritization process will be applied to that function as well.

CFG of the other function named onLocationChange() is shown in figure 5.4.

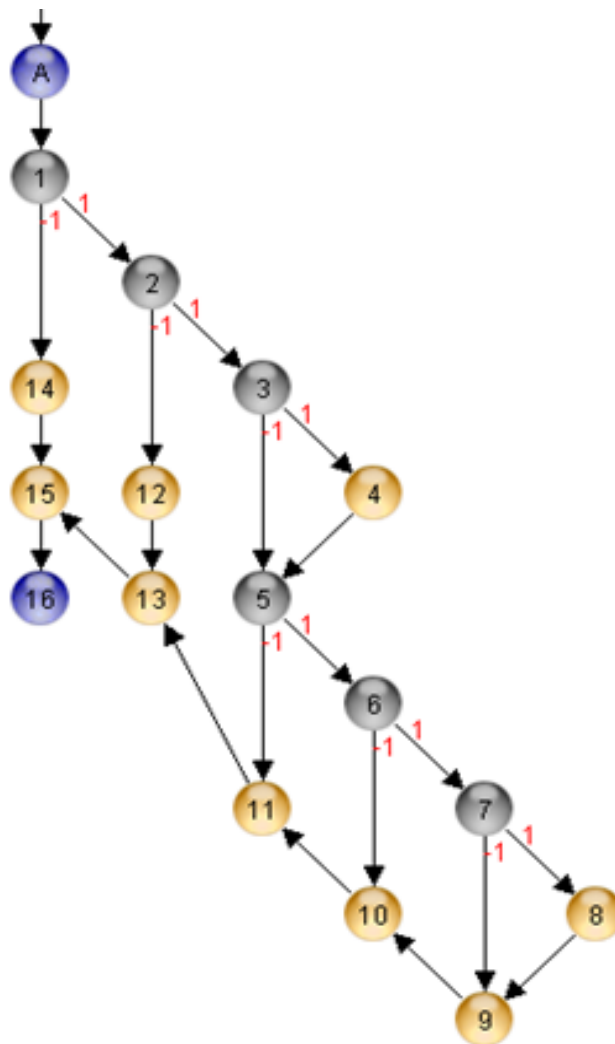


FIGURE 5.4: CFG of onLoctaionChange():AndroidRun Application

onLocationChange function of AndroidRun is statically analyzed and the nodes are annotated as follows:

- GPS is acquired at node 4 and node 12
- but GPS is not used or released

Figure 5.5 shows the test paths of the CFG shown in figure 5.4. These test paths are generated using the node coverage criteria through AutoTester Tool, based on the CFG shown in figure 5.4. Four test paths are generated by the tool.

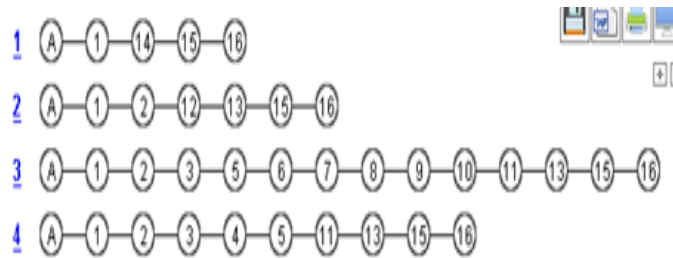


FIGURE 5.5: Test Paths of onLocationChange():AndroidRun Application

Table 5.10 includes each test path's statement coverage and energy bug coverage w.r.t resource being used.

TABLE 5.10: Coverage Information of Test Paths of onLocationChange(): AndroidRun Application

Test Paths	Energy Covered	Bug	No of Bugs	Resource Used	Statement Covered
TP1	No Bug	0	N/a	1, 14, 15, 16	
TP2	No Bug	0	N/a	1, 2, 12, 13, 15, 16	
TP3	AcqBug	1	GPS	1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 13, 15, 16	
TP4	AcqBug	1	GPS	1, 2, 3, 4, 5, 11, 13, 15, 16	

TABLE 5.11: onLocationChange(): Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage:Step-I

Test Path No	Energy Bug Weight	Normalized Energy Bug Weight	Statement Coverage	Normalized Statement Coverage	Test Path Weight
TP1	0	0	4	0	0
TP2	0	0	6	0.2	0.1
TP3	9	1	14	1	1
TP4	9	1	9	0.5	0.75

From the table 5.11, shows the test path's normalized weights and cumulative weight. Based on the test path weight; TP3 is selected as the weight of the test path is the highest among all.

$\langle \text{PrT} \rangle = \{ \text{TP3} \}$

TABLE 5.12: onLocationChange(): Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-StepII

Test Path No	Additional Energy Bug Weight	Normalized Energy Bug Weight	Additional Statement Coverage	Normalized Statement Coverage	Test Path Weight
TP1	0	0	1	1	0.5
TP2	0	0	1	1	0.5
TP4	9	1	1	1	1

Table 5.12 shows the additional coverage information after selecting TP3. As the weight of TP4 is the highest among all, so TP4 is selected.

$\langle \text{PrT} \rangle = \{ \text{TP3}, \text{TP4} \}$

TABLE 5.13: onLocationChange(): Test Path Weight Calculation from Normalized Energy Bug Weight and Statement Coverage-StepIII

Test Path No	Additional Energy Bug Weight	Normalized Energy Bug Weight	Additional State-ment Coverage	Normalized Statement Coverage	Test Path Weight
TP1	0	0	1	1	0.5
TP2	0	0	1	1	0.5

As both of the remaining test paths have the similar weights as shown in table 5.13 therefore, TP1 will be selected after TP4..

$$\langle \text{PrT} \rangle = \{ \text{TP3}, \text{TP4}, \text{TP1} \}$$

And then the last test path will be appended at the end. So, the prioritized list will be

$$\langle \text{PrT} \rangle = \{ \text{TP3}, \text{TP4}, \text{TP1}, \text{TP2} \}$$

5.3 Evaluation Parameters

For any approach presented in the domain of test case prioritization, it is essential to evaluate their effectiveness by performing metric measurements. Therefore, evaluation metric is significant to measure the efficiency of any test case prioritization approach. Figure 5.6 depicts the widely utilized evaluation metrics being used in this domain [58].

Figure shows that the Average Percentage Fault Detection (APFD) metric has been widely used for evaluating TCP techniques whereas execution time is the least used metric for this purpose. While using the APFD as evaluation metric, it has been assumed that each test path incurs the same execution time and cost [34].

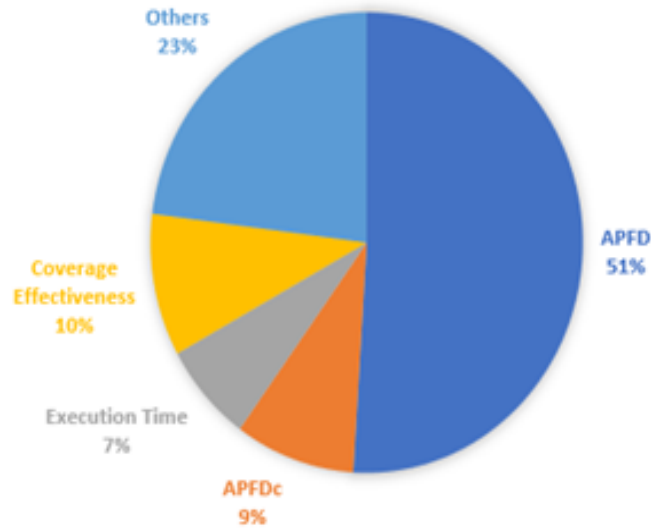


FIGURE 5.6: TCP Evaluation Metric Types [58]

Formula to calculate the APFD value is defined as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + \dots + TF_n}{nm} + \frac{1}{2n} \quad (5.1)$$

Where n is the number of test cases, m represents the number of faults and TF_i is the order of the first test case that reveals the i th fault F_i .

To access the proposed prioritization approach, we have used two variants of APFD metric, which is APFD-Bug variant and APFD-Statement Coverage variant.

5.3.1 APFD-Bug Variant

In the bug variant metric of APFD, the energy bugs are considered as the faults, which will be calculated using the following equation.

$$APFD_{Bug} = 1 - \frac{TB_1 + TB_2 + TB_3 + \dots + TB_n}{nm} + \frac{1}{2n} \quad (5.2)$$

Where T is the test suite containing n test cases and B is the set of m energy bugs revealed by T . For ordering T' , TB_i is the order of the first test case that reveals the i th energy bug B_i .

5.3.2 APFD-Statement Coverage Variant

In the statement coverage variant metric of APFD, the faults are seeded in some lines of code which are considered as the faults, which will be calculated using the following equation.

$$APFD_{StC} = 1 - \frac{TSC_1 + TSC_2 + TSC_3 + \dots + TSC_n}{nm} + \frac{1}{2n} \quad (5.3)$$

Where T is the test suite containing n test cases and SC is the set of m statement level faults revealed by T. For ordering T', TSC_i is the order of the first test case that reveals the ith statement fault SC_i .

5.4 Results

We have used 10 different open-source Android application available on GitHub. The selected applications use different resources like GPS, Wi-Fi, Bluetooth and wakeLock.

TABLE 5.14: Application's $APFD_{Bug}$, $APFD_{StC}$ and $APFD_{(B+St)}$ values at $\alpha = 0.5$

Sr No	Applications	$APFD_{Bug}$	$APFD_{StC}$	$APFD_{(B+St)}$
1	AndroinRun	0.83	0.77	0.78
2	Jamendo	0.8	0.9	0.87
3	Open Camera	0.81	0.76	0.77
4	A2DP volume	0.84	0.73	0.72
5	Apollo	0.75	0.75	0.75
6	Andromatic	0.79	0.79	0.73
7	K9 Mail	0.86	0.72	0.75
8	MyTracks	0.79	0.67	0.66
9	Find3	0.69	0.621	0.63
10	WiFiGPS Loca- tion	0.8	0.7	0.78

Initially the experiment is performed keeping the alpha value as 0.5. Table 5.14 shows the $APFD_{Bug}$, $APFD_{StC}$ and $APFD_{(B+St)}$ values.

$APFD_{Bug}$, $APFD_{StC}$ and $APFD_{(B+St)}$ value for each application has been calculated for each application. $APFD_{(B+St)}$ has been calculated considering both energy bugs and the statement level faults at the same time, shown in table 5.14.

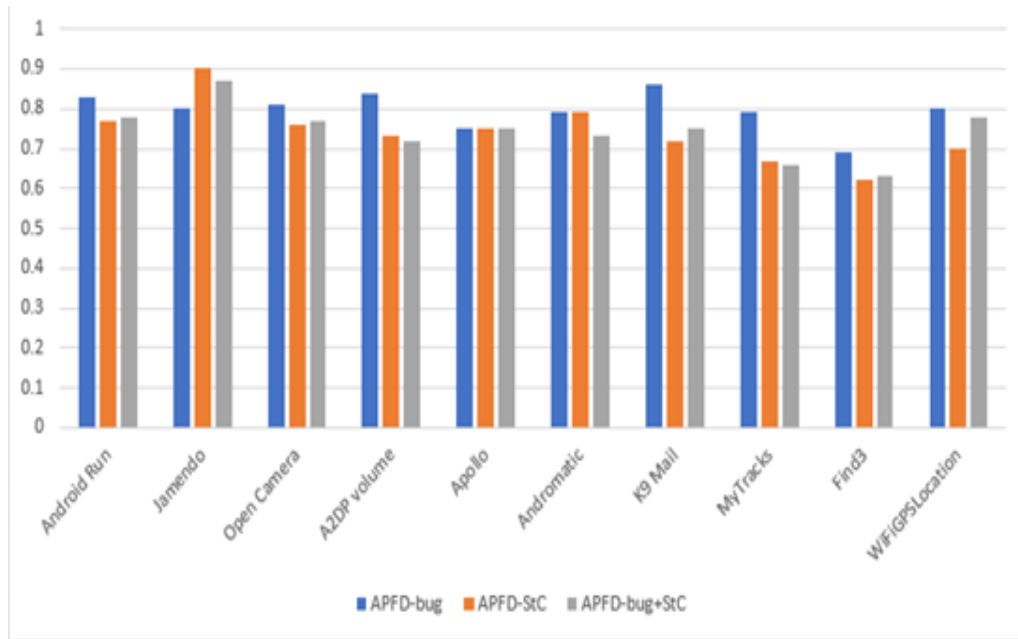


FIGURE 5.7: Application's $APFD_{Bug}$, $APFD_{StC}$ and $APFD_{(B+St)}$ values

Figure 5.7 shows that the energy bug revealing capability of the proposed approach is between 70-80%. The graph shows that in the majority of the applications, the proposed approach detects 80% of the energy bugs. While in two applications, Apollo and Find3, the results are less than 80% ($80\% < \text{but} > 70\%$).

75 to 85% of the statement level faults are uncovered using the proposed TCP technique. Whereas in Jamendo (application) 90% of the statement level faults are revealed while in MyTracks and Find3, the ratio remains between 63 to 70%.

However, considering both types of faults at a time, the results are somewhat similar to the $APFD_{Bug}$. Higher fault revealing percentage can be seen in Jamendo (approx 87%). Find3 and MyTracks show less fault coverage percentage, which is between 60 to 70%.

5.5 Comparison

For comparing our approach, we have selected energy-aware test suite minimization [38]. The paper presents two approaches that identifies energy intensive segments of the code and performs test suite minimization.

TABLE 5.15: Comparison of Proposed Approach with Existing Approach

Comparison Factors	Energy aware Test-Suite Minimization [38]	Proposed Energy aware Prioritization Approach
Granularity Level	Method Level	Statement Level
Code Coverage	No	Yes
Energy Bug Coverage	Yes	Yes
Prioritization Vs Minimization	Minimization is used, which removes the redundant test cases	Prioritization algorithm is applied which does not discards any of the test case
Energy Cost Estimation	Executes the test paths and based on method invocation and System APIs	Energy based weights are calculated without executing the test paths

Table 5.15 shows the comparison between proposed approach and the approach presented in [38], these points are further elaborated as follows:

- **Approach Granularity:** The approach presented in the selected paper is the method level approach which covers energy-greedy segments of the

code. Test paths covering more energy greedy segments is considered as the most energy intensive path of the application. But considering coarse grained granularity level does not ensures the statement level coverage of the application. As the method may have many paths inside it, therefore executing a method does not ensure that all the paths of the method are executed.

Our approach overcomes this loop hole by presenting fine grained approach that works on the statement level. The approach will be able to detect the bug-prone paths inside the methods. Hence providing both the energy coverage and the statement level coverage.

- **Code Coverage:** Method level approach does not ensure code coverage as it does not cover all the statements inside a method. Therefore, the approach presented in [38] only focuses on the energy intensive paths not on the functionality bugs.

Keeping this point in consideration, we presented an approach that will be able to provide the code coverage along with the energy bug coverage. The proposed approach considers statement coverage as well as energy bug coverage while calculating the weights of the test paths. Statement coverage has been given equal importance while calculating the weights hence makes the technique more efficient in detecting functionality bugs as well as energy bugs.

- **Energy Bugs:** Despite of calculating energy consumption of the segments, our approach focuses on the energy bugs that can appear in the code segments making it consumes more energy while execution. The minimization approach is not able to uncover the possible energy bugs which may occur while execution.
- **Prioritization vs Minimization:** Test Suite minimization approach tends to remove redundant test cases using some criteria in order to reduce the size of the test suite. The redundant test cases are removed permanently,

however empirical evidences proves that test suite's fault detection capability can be compromised by minimization [2]. Additionally, any test case can be considered as redundant with one specific criterion but is not redundant with reference to any other criteria. But test case prioritization schedules the test cases in order to increase their effectiveness. As the prioritization technique doesn't discards any test case therefore the drawbacks of the minimization approach can be avoided by preferring TCP over TSM.

- **Energy Cost Estimation:** The paper presented Integer Linear Programming based approach for TSM which executes the test paths to detect energy intensive code segments (i.e. methods) based on method invocation and System APIs. Executing the test paths makes it more costly (in terms of time and resource utilization required for execution). Our approach presents greedy approach for prioritizing the test suites. For prioritizing the test suite, energy based weights are calculated without executing the test paths.

Chapter 6

Conclusion and Future Work

Now-a-days applications may undergo several changes for incorporating news requirements or rectifying the bugs. Therefore, regression testing is required where there is need to test the newly built version of that software. Additionally, with the growth in size and complexity of mobile applications energy bugs become an important factor to be considered. Keeping these factors in consideration energy aware testing and functional testing needs to be performed in parallel for testing the mobile applications.

In this research, we have proposed an energy aware test case prioritization approach for Android applications. The proposed technique works on statement level and is able to detect energy bugs as well as the functionality bugs. The existing techniques work on method level which only cover energy greedy segments of the code and unable to detect code-level energy bugs and functionality bugs. Any path in the CFG (Control Flow Graph) which acquires or uses a resource but does not release it at the end is considered as a bug-prone path. The proposed approach assigns weights to each test paths based on their energy bug and statement coverage. Using the assigned weights, the test case prioritization is performed. Results shows that the proposed technique is able to detect 72 to 87% of the energy bugs. Similar percentage is observed in detecting both the functionality and energy bugs collectively. but two applications (MyTracks and Find3) show some variation by detecting giving 60-70% of the bugs.

Extending our approach, several other directions can be studied in future. The work can be extended to consider energy hotspots at the code level for assigning weights of test paths. While calculating weights w.r.t energy hotspots the challenging factor is to calculate the weights based on tail energy hotspot. Tail energy hotspot is basically the energy consumed by acquiring any resource too early or releasing it too late. Similarly, the approach can be extended to the application level by detecting the energy and functional bugs collectively. For considering energy bug at application level, test paths from Event Flow Graph (EFG) need to be generated and then the proposed prioritization criteria will be applied to obtain prioritized test cases. Apart from that, other graph coverage criterion (i.e., branch, loop coverage etc.) can be applied and study the variations in results.

Bibliography

- [1] R. Verdecchia, A. Guldner, Y. Becker, and E. Kern, “Code-level energy hotspot localization via naive spectrum based testing,” in *Advances and New Trends in Environmental Informatics*, pp. 111–130, Springer, 2018.
- [2] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.
- [3] A. P. Mathur, *Foundations of software testing, 2/e*. Pearson Education India, 2013.
- [4] P. A. d. M. S. Neto, I. d. C. Machado, Y. C. Cavalcanti, E. S. d. Almeida, V. C. Garcia, and S. R. d. L. Meira, “A regression testing approach for software product lines architectures,” in *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*, pp. 41–50, 2010.
- [5] G. Rothermel and M. J. Harrold, “A framework for evaluating regression test selection techniques,” in *Proceedings of 16th International Conference on Software Engineering*, pp. 201–210, IEEE, 1994.
- [6] M. R. Garey, “A guide to the theory of np-completeness,” *Computers and intractability*, 1979.
- [7] P. R. Srivastava, “Test case prioritization.,” *Journal of Theoretical & Applied Information Technology*, vol. 4, no. 3, 2008.
- [8] W. Sun, Z. Gao, W. Yang, C. Fang, and Z. Chen, “Multi-objective test case prioritization for gui applications,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1074–1079, 2013.

-
- [9] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," *Journal of King Saud University-Computer and Information Sciences*, 2018.
- [10] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.
- [11] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh, "An improved method for test case prioritization by incorporating historical test case data," *Science of Computer Programming*, vol. 78, no. 1, pp. 93–116, 2012.
- [12] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *Journal of Systems and Software*, vol. 85, no. 3, pp. 626–637, 2012.
- [13] R. U. Maheswari and D. J. Mala, "Heuristic-based time-aware multi-criteria test case prioritisation technique," *International Journal of Information Systems and Change Management*, vol. 9, no. 4, pp. 315–333, 2017.
- [14] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements based test prioritization using risk factors: An industrial study," *Information and Software Technology*, vol. 69, pp. 71–83, 2016.
- [15] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, pp. 465–474, Ieee, 2007.
- [16] M. Mahdieh, S.-H. Mirian-Hosseiniabadi, K. Etemadi, A. Nosrati, and S. Jalali, "Incorporating fault-proneness estimations into coverage-based test case prioritization methods," *Information and Software Technology*, vol. 121, p. 106269, 2020.
- [17] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 523–534, IEEE, 2016.

- [18] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 235–245, 2014.
- [19] O. Dahiya and K. Solanki, “A systematic literature study of regression test case prioritization approaches,” *International Journal of Engineering & Technology*, vol. 7, no. 4, pp. 2184–2191, 2018.
- [20] B. Qu, C. Nie, B. Xu, and X. Zhang, “Test case prioritization for black box testing,” in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, pp. 465–474, Ieee, 2007.
- [21] S. Mohanty, A. A. Acharya, and D. P. Mohapatra, “A survey on model based test case prioritization,” *International Journal of Computer Science and Information Technologies*, vol. 2, no. 3, pp. 1042–1047, 2011.
- [22] M. Wan, Y. Jin, D. Li, J. Gui, S. Mahajan, and W. G. Halfond, “Detecting display energy hotspots in android apps,” *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1635, 2017.
- [23] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, “Detecting energy bugs and hotspots in mobile apps,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 588–598, 2014.
- [24] D. Li, S. Hao, J. Gui, and W. G. Halfond, “An empirical study of the energy consumption of android applications,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 121–130, IEEE, 2014.
- [25] V. Kudva, *Fault Driven Supervised Tie Breaking for Test Case Prioritization*. PhD thesis, University of Waterloo, 2018.
- [26] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

-
- [27] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, “A unified test case prioritization approach,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–31, 2014.
- [28] A. Ammar, S. Baharom, A. A. Abd Ghani, and J. Din, “Enhanced weighted method for test case prioritization in regression testing using unique priority value,” in *2016 International Conference on Information Science and Security (ICISS)*, pp. 1–6, IEEE, 2016.
- [29] S. Wang, J. Nam, and L. Tan, “Qtep: quality-aware test case prioritization,” in *Proceedings of the 2017 11th Joint Meeting on foundations of software engineering*, pp. 523–534, 2017.
- [30] I. Alazzam and K. M. O. Nahar, “Combined source code approach for test case prioritization,” in *Proceedings of the 2018 International Conference on Information Science and System*, pp. 12–15, 2018.
- [31] W. N. Torres, E. L. Alves, and P. D. Machado, “An empirical study on the spreading of fault revealing test cases in prioritized suites,” in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 129–138, IEEE, 2019.
- [32] T. Afzal, A. Nadeem, M. Sindhu, and Q. uz Zaman, “Test case prioritization based on path complexity,” in *2019 International Conference on Frontiers of Information Technology (FIT)*, pp. 363–3635, IEEE, 2019.
- [33] A. Ammar, A. A. A. G. Salmi Baharom, and J. Din, “The effectiveness of an enhanced weighted method with a unique priority value for test case prioritization in regression testing,” *International Journal of Engineering & Technology*, vol. 7, no. 4.31, pp. 20–27, 2018.
- [34] R. Huang, Q. Zhang, D. Towey, W. Sun, and J. Chen, “Regression test case prioritization by code combinations coverage,” *Journal of Systems and Software*, vol. 169, p. 110712, 2020.

- [35] Y. ShAo, B. Liu, S. WAng, and P. XiAo, “A novel test case prioritization method based on problems of numerical software code statement defect prediction nowatorska metoda priorytetyzacji przypadków testowych oparta na prognozowaniu błędów instrukcji kodu oprogramowania numerycznego,” *EK-SPLOATACJA I NIEZAWODNOSC*, vol. 22, no. 3, p. 419, 2020.
- [36] D. Li, C. Sahin, J. Clause, and W. G. Halfond, “Energy-directed test suite optimization,” in *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, pp. 62–69, IEEE Computer Society, 2013.
- [37] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. Halfond, “Integrated energy-directed test suite optimization,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 339–350, 2014.
- [38] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, “Energy-aware test-suite minimization for android apps,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 425–436, 2016.
- [39] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, “Multi-objective optimization of energy consumption of guis in android apps,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, pp. 1–47, 2018.
- [40] D. Li, S. Hao, W. G. Halfond, and R. Govindan, “Calculating source line level energy information for android applications,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 78–89, 2013.
- [41] M. Wan, Y. Jin, D. Li, J. Gui, S. Mahajan, and W. G. Halfond, “Detecting display energy hotspots in android apps,” *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1635, 2017.
- [42] M. U. Farooq, S. U. R. Khan, and M. O. Beg, “Melta: A method level energy estimation technique for android development,” in *2019 International Conference on Innovative Computing (ICIC)*, pp. 1–10, IEEE, 2019.
- [43] M. Couto, J. Saraiva, and J. P. Fernandes, “Energy refactorings for android in the large and in the wild,” in *2020 IEEE 27th International Conference*

- on Software Analysis, Evolution and Reengineering (SANER)*, pp. 217–228, IEEE, 2020.
- [44] L. Al Shalabi and Z. Shaaban, “Normalization as a preprocessing engine for data mining and the approach of preference matrix,” in *2006 International conference on dependability of computer systems*, pp. 207–214, IEEE, 2006.
- [45] M. Javeed, “Tester v1.0.”
- [46] L. Corral, A. B. Georgiev, A. Sillitti, and G. Succi, “A method for characterizing energy consumption in android smartphones,” in *2013 2nd international workshop on green and sustainable software (GREENS)*, pp. 38–45, IEEE, 2013.
- [47] “Where the world builds software.”
- [48] Mattkirwan, “mattkirwan/androidrun.”
- [49] Telecapoland, “telecapoland/jamendo-android.”
- [50] Almalence, “almalence/opencamera.”
- [51] RoosterRat, “Roosterrat/a2dpvolume.”
- [52] Ctripcorp, “ctripcorp/apollo.”
- [53] FleetC0m, “Fleetc0m/andromatic.”
- [54] jca02266, “jca02266/k9mail.”
- [55] Plonk42, “Plonk42/mytracks.”
- [56] Schollz, “schollz/find3-android-scanner.”
- [57] Falaki, “falaki/wifigpslocation.”
- [58] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, “Test case prioritization approaches in regression testing: A systematic literature review,” *Information and Software Technology*, vol. 93, pp. 74–93, 2018.