Test Case Prioritization for Mobile Apps

1st Zeinab Abdalla
Department of Computer Science
University of Denver
Denver, USA
Zeinab.Abdalla@du.edu

2rd Kerstin Haring
Department of Computer Science
University of Denver
Denver, USA
Kerstin.Haring@du.edu

3nd Anneliese Andrews
Department of Computer Science
University of Denver
Denver, USA
Anneliese.Andrews@du.edu

***Abstract*—Like for any software, Mobile Applications (Apps) are modified during their specification, implementation, and maintenance phases with the goal to satisfy new requirements, fix defects, and change or add functionality. There is a need to regression test for and detect faults in every phase. However, resource and time constraints may lead to Mobile Apps not being tested. In this paper we present a model-based test approach to prioritize test cases based on the input complexity for each test path of the Mobile App. We argue that this novel approach will significantly improve the efficiency and effectiveness of current techniques.**

***Index Terms*—Model-Based Regression Testing, Test Case Prioritization, Mobile Apps**

I. Introduction

With the growing number of Mobile Applications and the increasing functionality and complexity of the applications, developers change and evolve their applications frequently. The frequent changes in every step of the development process and maintenance process highlight that there is an increased need for efficient software regression testing approaches [17]. Regression testing is essential to make sure that the new introduced changes do not impact the behaviors of the existing, unchanged part of the Mobile App under test. However, regression testing has resource and time constraints, and that in turn may increase the need to detect faults in the modified Mobile App as early as possible. Test Case Prioritization (TCP) limits these constraints by ordering test cases in a way that maximizes early fault detection in a software system [2]. Test case prioritization techniques schedule test cases for execution, so it increases the rate of fault detection and provides faster feedback on the system under test (SUT). This results in faults to be detected and corrected earlier. TCP is therefore an effective and and practical technique that helps to increase the rate of fault detection as software evolves. The test case prioritization problem has been formally defined as follows [1]: Given a test suite T, the set of permutations PT of T, and a function f from PT to the real numbers $\mathbb{R}$. The problem is then defined as to finding $T' \in PT$ such that $(\forall\ T'')$ where $(T'' \in PT)$ and $(T'' \neq T')$ such that $[f(T') \geq f(T'')]$. The set of permutations PT represents the set of all possible orderings of T, and f is a function that, when applied utilized to any such ordering, produces an award value for that ordering. Without loss of generality, this definition assumes that higher award values are preferable to lower ones.

Test case prioritization (TCP) techniques aim to improve regression testing cost effectiveness by revealing faults early in testing [1]. Test cases are ranked or prioritized to achieve required goals. The simplest method for test prioritization is random test prioritization where the test cases are ordered randomly. However, for a test suite of size N there are $N!$ possible test paths. Random prioritization selects randomly one of these paths. As factorial functions grow by multiplying by an increasing amount, the likelihood of a random selection of an appropriate test path is very low for even small test suites. We therefore focus in this paper on test case prioritization for black-box model-based regression testing which enables us to choose more appropriate test paths. Traditionally, the testing of any systems is modeled as finite state machine (FSM) which are derived from application requirements. However, testing mobile applications with FSMApp has not addressed test case prioritization. We expect that this novel approach will lead to more efficient and effective testing of mobile Apps.

Our work presents a model-based approach to prioritize test cases for Mobile Apps using FSMApp. Test cases are prioritized based on the input complexity for each test path of the Mobile App. The goal of model-based test prioritization is to detect faults in a new or modified Mobile App software early on to produce high quality software and increase efficiency of Mobile App development.

The paper is organized as follows: Section II describes existing work related to model-based test case prioritization, regression testing of mobile apps. Section III introduces the test case prioritization process. An example illustrates how prioritize test cases based on input complexity. Section IV draws conclusions and suggests further work.

II. Background

Several methodologies for model-based test case prioritization have been proposed and used. For example, Huang et al. [2] design and analyze a Graphical User Interface (GUI) test case prioritization approach with weights. They assign weights for each input and each action, resulting in a scoring system for test cases. The more important inputs have a higher weight. They proposed three methods for assigning a weight to each input: equal weight, fault-prone weight, and random weight. They used weight-based Event Flow Graph (EFG) methods to prioritize the test cases.

2769-5654/22/$31.00 ©2022 IEEE
DOI 10.1109/CSCI58124.2022.00332

Korel et al. [3] used a different method. They introduce methods of test prioritization based on extended finite state machines (EFSM) after changes to the model and the system. They present an analytical framework for evaluation of test prioritization methods. In addition Korel et al. [4], [5] present a model-based prioritization for modifications where models are not modified, only the source code is modified. The method is based on identifying elements of the model related to source-code modifications and to collect information during analysis of a model to use them to prioritize tests for execution. They present five heuristic model-based test prioritization methods that prioritize test cases based on dependency.

Sapna et al. [6], [7] proposed a prioritization technique based on Unified Modeling Language (UML). The constructs of an activity diagram are used for test paths prioritization. In [6] activity diagrams represent the scenarios of the system under test. A scenario is a complete path through the activity diagram. They prioritize test paths by assign weights to nodes and to edges then calculate weight of path. They assign weights to each of these nodes based on the possibility of occurrence of defects. And they assign weights to each of these edges based on the number of incoming dependencies of node and outgoing dependencies of node. Their approach [7] is to prioritizing use cases from UML use case diagrams. The developer assigns the priority of each actor on a scale of 0-10. The customer assigns the priority of each actor. Customers rate uses cases as very low, low, medium, high, very high (scale 1-10). Finally, calculate use case priority.

Amalfitano et al. [10] use a crawler-based technique to generate tests for Android apps. The crawler builds an event-based GUI tree and generate tests from this tree. They rerun previously tests and check whether program behavior has changed. This check is based on comparing sequences of user interface obtained in both test runs.

Do et al. [14] provide an approach for limited selective regression testing of Android apps based on code impact analysis and coverage information from rerunning the original test suite.

Chang et al. [12] emphasise repairing of obsolete test scripts for Android Apps. They build a model of the possible GUI event sequences semi automatically. When changes to screens, widgets or events occur, they are extracted by analyzing changes to the binary files. They use test simulation to identify which screens and widgets a test covers. This helps in identifying ranges within test scripts that need to be repaired.

Jiang et al. [13] aim to improve safety of regression test selection. Their approach concentrates on models of asynchronous task invocations, but also considers native code. This approach uses impact analysis based on a inter procedural control flow graph that models what is considered dangerous edges. Test case selection uses edge coverage information for each of the original tests. Tests that cover edges flagged as dangerous by the impact analysis are selected.

Alhaddad et al. [17] extend the FSMWeb [15] approach for testing web applications to the FSMApp approach for testing mobile applications. Then we extend it to FSMApp selective regression testing [8], and FSMApp test case minimization for Mobile apps [9]. Our paper aims to extend it to a test case prioritization for FSMApp.

Abdalla et al. [8] proposed selective regression testing for mobile application. They adapt the FSMWeb approach [11] for selective regression testing for mobile applications. They applied rules to classify the original set of tests into obsolete, retestable, and reusable tests based on the types of changes to the model. New tests are added to cover portions the App that have not been tested.

Abdalla et al. [9] present a test case minimization of FSMApp method for mobile Apps. It is based on concept analysis that removes redundant test cases. FSMApp generates a test suite T that satisfies test requirements based on graph coverage criteria and test aggregation criteria for its hierarchical model. Together, these constitute a set of test requirements R. Based on the relationship between T and TR, a concept lattice is created which is subsequently used to minimize the test suite T.

III. Test Case Prioritization

A. Test Case Prioritization Process

The idea of our model-based test prioritization approach is to use the original model and the modified model to identify a difference between these models. The collected information is then used to prioritize the test suite. We suggest to prioritize tests based on input complexity, since more inputs might be associated with a more complex functionality which in turn would make it more fault prone. Figure 1 shows test case Prioritization of FSMApp Process. We prioritize the test suite based on the number of inputs (complexity) for each test path. Identifying which test path that has biggest number of inputs (values and actions) and give it high priority. Then ordering the test paths in descending order, so the test path (with more complexity of inputs) that has highest priority will execute first. Our approach can be used to prioritize all the original test suite in rerun testing approaches and can be used to prioritize subset of tests that is selected from existing test suite.

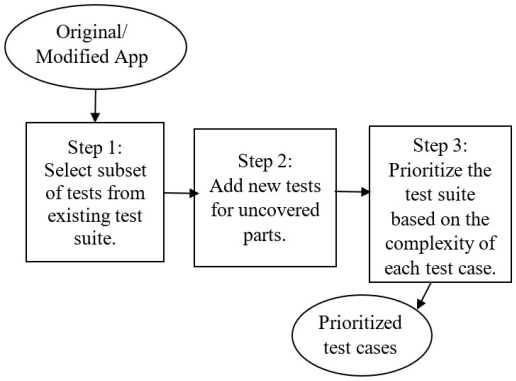

Fig. 1. Test Case Prioritization of FSMApp Process

As shown in figure 1 the process has three main steps. The first step selects a subset of tests from the existing test suite, the second step adds new tests, and the third step prioritizes the test suite. In more details, the first step is carried out and achieved by the following sub-steps.

- Build a hierarchical model HFSM' for the modified mobile App as proposed in [8].
  - Partition the mobile App into clusters.
  - Define logical App pages (LAP) and input constraints.
  - Build FSM for the clusters.
- Identify a difference between the original model HFSM and the modified model HFSM′.
- Select tests related to the modified parts of the model from existing test suite.

The second step adds new tests for uncovered parts. This is achieved by generating new abstract test cases for uncovered parts. The third and last step prioritizes the test suite based on the complexity of each test case. This is achieved by:

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

B. Example Used to Illustrate Approach

We used To Do App example to illustrate our approach of FSMApp test case prioritization. To Do is a simple app to do lists of tasks. The app has these services: add tasks, modify task (delete task, edit task, mark for a task done or undo), and search for a task. We modified the app to use it to explain our approach. More details about the example see [8], [9]. Figure 2 shows the original model HFSM for the ToDo App. Table I shows the original test suite for the ToDo App.

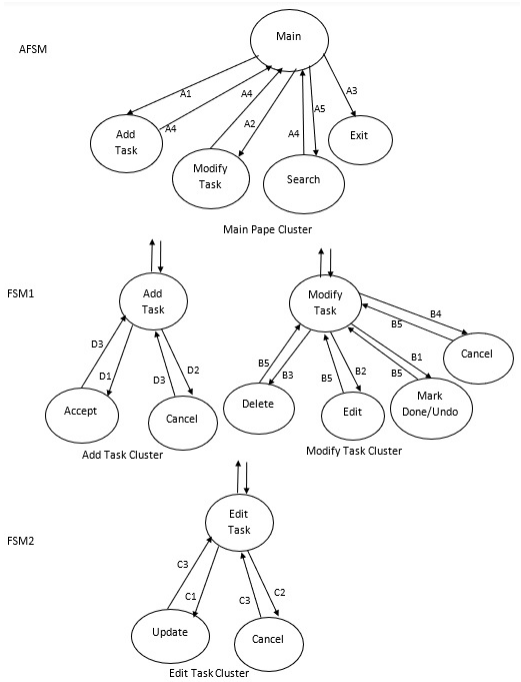

Fig. 2. Original Model HFSM for ToDo App

TABLE I
THE ORIGINAL TEST PATHS OF THE TO DO APP

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

Now, we need to apply our process to prioritize test cases for modified version of the ToDo App.

1) Step1: Select subset of tests from existing test suite: Build a hierarchical model HFSM' for the modified mobile App: We used FSMApp approach [8], [17] to build the hierarchical model HFSM' for the modified ToDo App. We Partitioned the ToDo mobile App into clusters, defined the logical App pages (LAP) and input constraints for it, then built FSM for the clusters. The constraints on inputs for ToDo App are:

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

We explained the input constraints and input types for the ToDo App in details in [8]. Table II shows input types for the ToDo App. Figure 3 shows the modified hierarchical model for ToDo Mobile App.

TABLE II
THE INPUT TYPES OF THE TO DO APP

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

Then we identify the difference between the original model HFSM and the modified model HFSM': Here we used the original model HFSM for ToDoApp in figure 2 and the modified model HFSM' in figure 3 to identify a difference between these models, and determine the modified parts. So can determine test paths related to them.

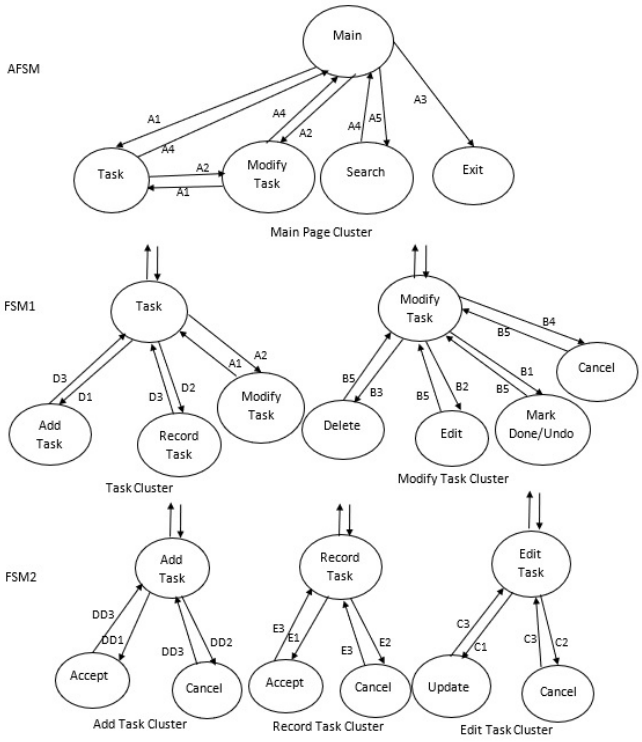

Fig. 3. Modified Model HFSM' for ToDo App

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

1) Node deletion, edge/node additions: Add Task node is deleted. Then a new cluster node is added, that gave options to add new tasks by recording information of the task and by writing the information of the task. Also, modify task by editing information, delete task, and mark done or undo of the task. Delete Add Task node, adding the new cluster node (Task), and adding a new FSM for the Task cluster caused that the abstract test cases associated with test path 1 in Table I effected paths.
2) Edges additions: Adding edges between cluster node Task and cluster node Modify Task. These edges to allow modifying tasks without returning to the main page of the App. This modification makes test paths 2 and 3 in Table I effected paths (PC).

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

2) Step2: Add new tests for uncovered parts: In this step, we determined if any parts of the system have not been tested and generated a new set of tests. We used a partial generation [16] to generate the test cases because the changes in the model are localized, details about generated the new tests in [8], [16]. Since the test case1 in Table I is not valid any more, we need new tests to cover untested parts. The following four new abstract test cases covered the paths of the model that are not covered by selected test cases.

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

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

TABLE III
THE SELECTED AND NEW TEST PATHS OF THE TO DO APP

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

3) Step3: Prioritize the test suite based on the complexity of each test case: In this step, we will prioritize test paths in Table III based on the complexity of each test path.

First, we will determine the complexity for each test path by determining the number of input values and actions. For example, Test Path1 has six inputs and five actions. The inputs are Task name (parname), Task Date and Time (ParD), (ParT)

which occurs twice in test path 1. The actions are click Add New Task button (b-ANT), Click Accept button(b-ANTA), Click Cancel (b-Cancel), and click back arrow to exit the mobile app (buttonBack). The complexity for this test path is $6 + 5 = 11$. This is calculated by:

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

where C is the total complexity for the test path, $InpValue$ are the input values, and $InpAction$ are the actions.

Tables IV to VII show the complexity for each test path. The first column shows transaction of the test path, the second column shows the constraint of the input, there third column shows the number of the input values, the fourth column shows number of the actions, and the fifth column swhos the complexity of each transaction for the test path.

TABLE IV
THE COMPLEXITY OF THE TEST PATH 1

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

TABLE V
THE COMPLEXITY OF THE TEST PATH 2

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

TABLE VI
THE COMPLEXITY OF THE TEST PATH 3

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

TABLE VII
THE COMPLEXITY OF THE TEST PATH 4

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

TABLE VIII
THE COMPLEXITY OF THE TEST PATH 5

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

TABLE IX
THE COMPLEXITY OF THE TEST PATH 6

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

The next step is to assign priority to each test case based on its complexity. Assign high priority to test path that has biggest number of inputs (values and actions).In this step, the test case that has high complexity is assigned with high priority. The priority start from 1 to N. One means to have the highest priority. The test path that has priority (1) will execute first. If there are paths that have the same priority, the tester randomly chooses which one execute first. Table X shows the priority for each test path.

TABLE X
THE PRIORITY OF THE TEST PATHS

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

Then we are ordering the test cases in descending order (high priority to low priority). The test case that has

high priority will be executed first. The prioritized test suite is:

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

.

IV. Conclusion

This paper presents a test case prioritization technique for mobile Apps. It describes a model-based black-box regression testing approach. The approach builds a hierarchical model for the modified mobile App, and uses the original model and the modified model to identify a difference between these models. The collected information is then used to select tests related to the modified parts of the model and new tests are added to cover portions that have not been tested. Then these test cases are prioritized based on the complexity of each of them.

Our technique can be used to improve the FSMApp [17] approach for testing mobile Apps by determining the input complexity of test paths in stage3 (select inputs) of their approach, and then prioritize the test paths based on the complexity for each test case before executing them. That means we can use our idea not just in terms of regression testing but also for the original versions of testing.

In our future work we plan to prioritize test cases based on weights of the inputs. The test path that has high weights of inputs has a high complexity, and will be assigned with a high priority. The test path that has the highest priority, meaning it has the highest weights of inputs and it is the highest complexity, so it will execute first. Basically, this idea works by assigning a weight value for each input and calculate the complexity of these weight value for each test case. The more important input has a higher weight value. For example, the importance of a menu-open action might be lower than a button click action because the former simply calls up a menu list, but the button click action might trigger one or more functions to change the application state. Therefore, according to the relationships between input types and errors, we will set different weight values for different input types. We will assume that higher-weighted actions will lead to higher rates of fault detection than lower-weighted inputs.

References

[1] G. Rothermel, RH. Untch, C. Chu, and MJ. Harrold, Prioritizing test cases for regression testing. IEEE Transactions on software engineering, 27(10), 929-948, 2001.

[2] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang: Design and analysis of GUI test-case prioritization using weight-based methods, Journal of System sand Software, vol. 83, no. 4, pp. 646–659, 2010.

[3] B. Korel, L. H. Tahat, and M. Harman: Test prioritization using system models, in 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 559–568.

[4] B. Korel, G. Koutsogiannakis, and L. H. Tahat: Application of system models in regression test suite prioritization, in 2008 IEEE International Conference on Software Maintenance, 2008, pp. 247–256.

[5] B. Korel, G. Koutsogiannakis, and L. H. Tahat: Model-based test prioritization heuristic methods and their evaluation, in Proceedings of the 3rd International Workshop on Advances in Model-Based Testing, ser. A-MOST '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 34–43.

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

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

[8] Z. Abdalla, A. Andrews, and A. Alhaddad. Regression testing of mobile apps. In 2021 International Conference on Computational Science and Computational Intelligence (CSCI), pages 1912–1917. IEEE, 2021.

[9] Z. Abdalla, A. Andrews, and A. Alhaddad. Test minimization for mobile app testing with fsmapp. In 2022 International Conference on Computer Science, Computer Engineering, Applied Computing (CSCE'22). Springer, 2022.

[10] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for android mobile application testing," in2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 252–261.

[11] A. A. Andrews, S. Azghandi, and O. Pilskalns, "Regression testing of web applications using FSMWeb," in Proceedings IASTED International Conference on Software Engineering and Applications, Nov. 2010.

[12] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, (2018, July). Change-based test script maintenance for android apps. In 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS) (pp. 215-225). IEEE.

[13] B. Jiang, Y. Wu, Y. Zhang, Z. Zhang, and W. K. Chan, (2018, July). ReTestDroid: towards safer regression test selection for android application. In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC) (Vol. 1, pp. 235-244). IEEE.

[14] Q. Do, G. Yang, M. Che, D. Hui, and J. Ridgeway, (2016, May). Regression test selection for android applications. In Proceedings of the International Conference on Mobile Software Engineering and Systems (pp. 27-28).

[15] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing web applications by modeling with FSMs," inSoftware and System Modeling, 2005, pp.326–345.

[16] A. Andrews and H. Do, "Trade-off analysis for selective versus brute-force regression testing in FSMWeb," in2014 IEEE 15th International Symposium on High-Assurance Systems Engineering, 2014, pp. 184–192.

[17] A. Alhaddad, A. Andrews, and Z. Abdalla. FSMApp: Testing Mobile Apps. In Advancing in Computers. Elsevier, 2021.