Developments in Automated Software Engineering: Automated Testing of Mobile Applications

Guduguntla Vamshi
NC State Univerisity
gudugu@ncsu.edu

Sattwik Pati
NC State Univerisity
spati2@ncsu.edu

ABSTRACT

This paper provides a comprehensive overview of the developments in the area of mobile application testing automation.For this purpose, we have reviewed the developments in papers published during 2008 to 2015.The line of research mainly includes focus on Event Sequence or test case generation,GUI testing, to evolutionary approach for system testing of apps.

The background of the study is the fact that mobile applications provide computational needs to millions of users. Hence, the reliability of the apps is so important as they use for financial services,business management,healthcare,etc domains.

We have discussed the motivation,all the related work, and techniques that were proposed in the papers, and used them to give a general idea of the progression of the research.

General Terms

Testing Automation,Scalability,Andriod,iOS

Keywords

Test Case generation, Adaptive random testing,Evodriod, Empirical bug studies

1. INTRODUCTION

Mobile app market brought about a fundamental shift in the way applications are served to the users. The pros include ability to quickly develop,deploy, maintain the apps used. This change however, has given rise to own set of problems.One of the authors - G Vamshi interned with Reveal Mobile Inc, a mobile audience data company based out of Raleigh witnessed the travails faced by a small organisation in planning the resources to fully test the products(apps,UI) before hitting the market.Besides, the shortcomings in the apps are exploited with malicious intent compromising the integrity of the apps on the market where they operate. For instance, Google Play which has 10 Mn + apps is a commonplace for security attacks on vulnerable apps. This situation is likely to get worse as apps are poised to be more complex keeping the demand for added functions in mind.

The testing of the mobile applications on mobile devices is difficult due to several reasons.Firstly, the mobile application has to deal with both user inputs and ever-changing environmental contexts. This makes it more complex for testers to generate test cases to expose software faults effectively. Secondly, the mobile device is resource limited.Thirdly, the diversity of mobile operating systems makes the programming style of different mobile devices differs significantly.

The state-of-practice in automated system testing of apps is random testing. Monkey[15] for Andriod is the tools which is de-facto industry standard that generates purely random tests. It provides a brute-force mechanism that achieves a shallow code coverage.

Other techniques have been proposed to improve the existing method. They include a GUI crawling-based technique [15], concolic testing technique [20], efforts to improve the input generation system using Dynadriod [4], a grey-box approach for automating the testing process by W. Yang, M. R. Prasad, and T. Xie . One technique which is closely realted to the Dynodriod and uses the evolutionary framework to test mobile apps is EvoDroid [19]. EvoDroid overcomes a key shortcoming of using evolutionary techniques for system testing, i.e., the inability to pass on genetic makeup of good individuals in the search. It achieves significantly higher code coverage than existing Android testing tools.

2. RELATED WORK

2.1 Input Generation Techniques

There are broadly three kinds of approaches for generating inputs to mobile apps: fuzz testing, which generates random inputs to the app; systematic testing, which systematically tests the app by executing it symbolically; and model-based testing, which tests a model of the app.

2.1.1 Fuzz Testing

The Android platform includes a fuzz testing tool Monkey [15] that generates a sequence of random UI events to unmodified Android apps in a mobile device emulator. Recent work has applied Monkey to find GUI bugs [9] and security bugs [18] in apps. Fuzz testing is a black-box approach, it is easy to implement robustly, it is fully automatic, and it can efficiently generate a large number of simple inputs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NCSU 2015,Raleigh USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

But it is not suited for generating inputs that require human intelligence (e.g., constructing valid passwords, playing and winning a game, etc.) nor is it suited for generating highly specific inputs that control the appâĂŹs functionality, and it may generate highly redundant in- puts. Finally, Monkey only generates UI events, not system events. It would be challenging to randomly generate system events given the large space of possible such events and highly structured data that is associated with many of them.

2.1.2 Systematic Testing

Several recent efforts [20] [10][18] have applied symbolic execution [17][[12] to generate inputs to Android apps. Symbolic execution automatically partitions the domain of inputs such that each partition corresponds to a unique program behavior (e.g., execution of a unique program path). Thus, it avoids generating redundant inputs and can generate highly specific inputs, but it is difficult to scale due to the notorious path explosion problem. Moreover, symbolic execution is not black-box and requires heavily instrumenting the app in addition to the framework.

2.1.3 Model-based Testing

Model-based testing has been widely studied in testing GUI-based programs [5] [13] [23] using the GUITAR framework [2]. GUITAR has been applied to Android apps, IPhone apps, and web apps, among others. In general, model-based testing requires users to provide a model of the appâĂŹs GUI [22], though automated GUI model inference tools tailored to specific GUI frameworks exist as well [8]. One such tool in the Android platform, that Dynodroid also uses for observing relevant UI events, is Hierarchy Viewer [3]. Model-based testing harnesses human and framework knowledge to abstract the input space of a programâĂŹs GUI, and thus reduce redundancy and improve efficiency, but existing approaches predominantly target UI events as opposed to system events.

2.2 Testing Frameworks

Over the years, there are several frameworks proposed for this purpose.

Amalfitano et al. [7] described a crawling-based approach that leverages completely random inputs to generate unique test cases. In a subsequent work [8], they presented an automated Android GUI ripping approach where task lists are used to fire events on the interface to discover and exercise the GUI model. Hu and Neamtiu [9] presented a random approach for generating GUI tests that uses the Android Monkey to execute

Yang et al. described a grey-box model creation technique which is concerned with deriving models for testing of Android app. Jensen et al. [6] presented a system testing approach that combines symbolic execution with sequence generation. An attempt was made to find valid sequences and inputs to reach pre-specified target locations.Anand et al. [20] presented an approach based on concolic testing of a particular Android library to identify the valid GUI events using the pixel coordinates.Choi et al. [21] proposed a machine learning approach to improve the app models by exploring the states not encountered during manual testing.

There has also been a recent interest in using cloud computing to validate and verify software. TaaS is a testing framework that automates software testing as a service on the cloud. Cloud9 provides a cloud-based symbolic execution engine.

3. CHECKLISTS

All the techniques have been presented with the results and a base-line study to show the evolution of the methods proposed to address the problem of automation.Throughout the progression of research we've looked out for the techniques keeping the below points in the checklist.

1. Robustness:Testing system ability to handle most of the listed apps in the market.
2. Black-box: Forgoing the need of app dependencies and the ability to de-compile the app binaries.
3. Versatile: Testing by accessing the most important app functionality.
4. Automating: How much does the testing system reduced the manual effort ?
5. Efficiency: Generation of test cases which spans most of the input combinations.

4. SAMPLING PROCEDURES

The papers and techniques we have studied came off by reviewing the Memon et al.[8] paper on GUI Ripping tool. This was highly cited in the year 2011, and then we have worked our way back and forth - 4 years in and out. This way, we could trace the journey and line of research for automated testing of mobile applications.

In the process, we have reviewed other papers which were to be studied to gain a sufficient understanding of the key terms used in those papers selected. As we have mentioned earlier, one of us have worked with engineers in a small organization facing a dilemma of huge dilemma of pulling-in large work hours and then cognitive attention given to existing manual testing process. Give an knowledge-full approach, which would eliminate a lot of time and money, would enhance the testing process to benefit small and large organization equally. We started to review works publish in the time frame 2008-2014 with the last review culminating in Evodriod - an evolutionary testing algorithm, which is especially closely related to Prof.Menzies discourse in the Automated Software Engineering course.

While listing the papers on this topic,some of the papers were side-lined which were mostly dealing with desktop applications, others marked as reference material to the existing line of research.

This paper presents a concise summary of the area, along with technical terms associated with it.

5. RESULTS

5.1 GUI Ripper

Keywords:

- **GUI Tree**: An intuitive representaion of the structure extracted during ripping. The nodes of the tree represent the individual User-Interfaces(screens) in the App, while the edges between them show the transition as a hierarchy relationship. Each node encapsulates the state of User-Interface,its objects and its properties.

- **Ripping**: A dynamic process in which the Graphic-User-Interface(of software or app) is automatically traversed by opening all the windows(or screens) and then extract all the widgets' properties and values. This extracted information is verified and used by the test designer to generate multiple test cases. GUI ripping helps to understand both structural and execution behaviour of the GUI.

- **Testing Automation**: The process of using a software to conduct the execution of a suite of test cases on another software is called test automation. The testing software also compares the actual outcomes of a test case with the predicted outcomes.

Motivational Statements:

- Even though developed on a Java Platform, Android applications(developed on Android Development Framework) differs greatly from the Java client-server framework. This gap in understanding from the developer's standpoint leaves room for defect injection

- Thus, the chief motivation of Amalfitano et. al is to develop an automated testing software that is capable of testing Android applications via their GUI

In this section, we discuss the results of the test performed by Memon et al.[8] using GUI Ripper on Andriod apps. The tool AndroidRipper automatically and systematically traverse the appâĂŹs user interface, generating and executing test cases as new events are encountered.Test case generation is based on the automatic dynamic analysis of the GUI that is executed in order to find and fire events in the GUI.AndroidRipper is based on a configurable GUI analysis technique, performed by a GUI ripper whose behavior can be tuned, via some parameters.

The evaluation of AndroidRipper using the âĂIJWordPress for AndroidâĂİ application revealed four undocumented bugs, automatically detected in less than five hours. This datum shows the effectiveness of the technique in finding real bugs and its suitability for testing processes that need to be carried out in a short amount of time. The experimental data showed that the proposed technique is more effective in bug detection than the random testing technique implemented by Monkey. Below is the comparison in four runs in Table 1

The results of a single execution of Monkey (RM) whose time duration was 4.46 hours, hence, comparable with the duration of R3 and R2 sessions. In the considered execution, Monkey had to fire 45,000 events with the default value of event type statistical distribution. In this experiment Monkey found 3 crashes corresponding to bug B2 and reached 25.27 perc LOC coverage, being less effective than the ripping based testing sessions R3 and R2. Monkey results are reported in the RM column of Table 1.

The app crash bugs B1,B2,B3,B4 correspond to the various operations in defaults,rapid actions,refresh,user sharing the post.

5.2 Adaptive Random Testing

Keywords:

- **Adaptive Random Testing**: It is an improved version of Random testing for test case generation. ART realizes the fact that the input failure regions of an app cluster together. So, it tries to spread the test case generation as evenly as possible, to detect the first failure. ART is 40-50 pc more quicker than Random testing to detect the first failure of an app.

- **Test Case generation**: Test cases are the actions formed by a set of events. The set of events form a action to be tested on a mobile app so as to observe the output. The primary aim is to look for where apps malfuncion against a given test case.

- **Random Technique**: This technique generates random test cases for mobile app testing. It is a fully automatic,non-intrusive and takes a pure random approach in simulation events from a random pool.There is chance that new event generated might be closer to the one generated in the previous event sequence set.

Table 1: Bugs, Coverage and Cost Data of GUI Ripper

Desc	R1	R2	R3	RM
Crashes of Bug B1	-	4	4	-
Crashes of Bug B2	-	1	1	1
Crashes of Bug B3	-	1	1	-
Crashes of Bug B4	-	-	2	-
Total Bugs	0	3	4	1
Total Crashes	0		8	3
Time (hours)	0.2	4.88	4.58	4.46

Motivational Statements:

- In Black box view, inputs are user-based and context-based. Memon et al. proposed to model GUI applications by Event-flow graphs. Their model accepts a predefined set of user inputs and system-generated events which produces a deterministic output. This does not take into account the non-determinism caused by dynamic context-based inputs.

- Lack of cheap and effective techniques for test-case generation is also a strong motivation of the authors. The earlier practice of record-and-replay the test script was highly subjective.

In Black box view, inputs are user-based and context-based. Memon et al.[8] proposed to model GUI applications by Event-flow graphs. Their model accepts a pre-defined set of user inputs and system-generated events which produces a deterministic output. This does not take into account the non-determinism caused by dynamic context-based inputs. Having a lack of cheap and effective techniques for test-case generation, Zhifang et al. [24] have proposed Adaptive technique for sequence generation. This was indeed expensive and detrimental to scaling.The authors wanted to address the problem by an automated and adaptive approach of testing.

The authors applied random strategy and ART to test one application at a time. The tool exposes a fault if a software crash or a system no response error is triggered.The metric used in evaluation is the number of test cases required to detect the first failure. It reflects the fault detection ability of the generated test cases. The second one is the time used

Table 2: Comparison of time(s) to first fault between Adaptive and Random testing

App	ART	Random
Bluetooth	400	500
Contacts	200	850
SMS	350	450
Bluetalk	350	425
Dialer	300	800
Browser	350	700

to find the first fault. This second metric also takes the time used for test case generation into consideration. The results are shown in Table 2.

In this technique, the authors proposed the event sequence distance to measure the distance between the test cases of mobile applications. Adaptive test case are generated for mobile application in a black box, non-intrusive manner.

Compared with Random, Adaptive Random Testing can both reduce the number of test cases and the time needed to expose the first fault significantly.

5.3 Event-flow model for testing

Keywords:

- **Event Flow model**: The Event-flow model represents the all possible sequences of events and interactions that can be executed by on a Graphical User Interface. This is similar to the the data-flow model which represents all the possible definitions and uses of a memory location.
- **Event Flow graph**: It shows all the possible event executions paths with each vertix representing the event and the edge showing the transition of one event to another.
- **Test-Oracles**: A test oracle is a mechanism that determines whether a piece of software executed correctly for a test case. In testing, the actual output is compared with a presumably correct expected output. A test oracle may be manual or automated.

Motivational Statements:

- White et al. presented a different state-machine model for GUI test-case generation that partitions the state space into different machines based on user tasks. The test designer identifies a responsibility, i.e. a user task that can be performed with the GUI. This approach was successful at partitioning the GUI into manageable functional units that can be tested separately. Drawback: Large manual effort is in designing the FSM model for testing, especially when code is not available.
- Another approach is to to mimic novice usersâĂŹ inputs. This approach relies on an expert to first manually generate a sequence of GUI events for a given user task. A genetic algorithm technique is then used to modify and lengthen the sequence, thereby mimicking a novice user. The underlying assumption is that novice users often take indirect paths through GUIs, whereas expert users take shorter, more direct paths. Drawback:This technique requires a substantial amount of manual effort, and cannot be used to generate other types of test cases.
- Other tools used for GUI testing require programming each GUI test case. These tools include extensions of JUnit such as JFCUnit, Abbot, Pounder, and Jemmy Module. Drawback: The tester should manually program the event interactions to test, and also specify expected GUI behaviour.

The author presents one scalable event-flow model [14] to semi-automatically reverse-engineer the model from an implementation. Earlier work on model-based test-case generation, test-oracle creation, coverage evaluation, and regression testing is recast in terms of this model by defining event-space exploration strategies (ESESs) and creating an end-to-end GUI testing process.

As the event-flow model is not tied to a specific aspect of the GUI testing process, it is used to perform a wide variety of testing tasks by defining specialized model-based techniques called event-space exploration strategies (ESESs). These ESESs use the event-flow model in a number of ways to develop an end-to-end GUI testing process.

The author showed that the application of this model for research and practice in automated testing. More specifically, the author showed that:

1. once the event-flow model is created, it can be used to generate a large number of GUI test cases with very little cost and effort
2. automated tools, built around the event-flow model, greatly simplify both model building as well as many GUI testing tasks
3. the model and tools enable large experiments in GUI testing.

The contributions of this paper include the following.

1. The consolidation of several existing GUI testing models into one event-flow model and its separation from specific testing tasks
2. The enhancement of the general event-flow model with customized ESESs to perform specific GUI testing tasks, leading to a more general solution to the GUI testing.
3. The employment of three ESESs (for model checking, test-case generation, and test-oracle creation) to develop an end-to-end GUI testing process and the demonstration of this process on non-trivial GUI-based applications via a scenario.

These author using a couple of scenarios showed that the event-flow model can be used to quickly generate a large number of GUI test cases that are effective at detecting GUI faults. The model also promotes re-usability as once created, it can be quickly redeployed to generate additional test cases for the same or a modified version of the GUI.The event-flow model is scalable, i.e. the size of the model grows linearly with the number of events in the GUI.The biggest challenge that this model currently poses is controlling the size of the space of all possible event interactions.

The author has contributed to the open source repository into a packaged software called GUITAR[2]

5.4 GUI bugs discovery

Keywords:

- **Reliability**: In testing activities, one of the primary objectives is to obtain a life distribution that describes the times-to-failure of a component, sub-assembly, assembly or system. This analysis is based on the time of successful operation or time-to-failure data of the item (component), either under use conditions or from accelerated life tests.
- **Empirical bug studies**: The bug recording study of the most popular and most diversely used mobile apps since the Andriod OS came into existence. The studies are focused on the apps which have a long life time, a detailed bug history reported by the developers and users. The reports should consist of broad range of representative bugs.
- **Tracing**: Tracing is the process of logging the details of each test case into a trace file. In this paper, the tracing activity is carried out while the application is tested against the test cases in the Virtual Machine.Tracing captures three kinds of events: Graphical User Interface (GUI) based, method calls and exceptions.

Motivational Statements:

- A. Chaudhuri. Language-based security on android. In PLAS âĂŹ09 They wrote about Andriod Verification.They presented a formal study of Android security. Their work included reasoning about their data flow security properties.The approach is deployed at the OS level and focus on security.
- A. Kervinen, M. Maunumaa, T. Pa ÌĺLa ÌĺLkko ÌĺLnen, and M. Katara. Model-based testing through a gui. In Formal Approaches to Software Testing, volume 3997, pages 16âĂŞ31. 2006. They presented a formal model and architecture for testing concurrently running applications.
- X. Yuan and A. M. Memon. Generating event sequence-based test cases using gui runtime state feedback. IEEE Trans. on Software Engineering, 36:81âĂŞ95, 2010. They proposed a model-based approach for testing GUI-based applications which can generate test cases automatically using a structural event generation graph

In this paper an approach for automating the testing process for Android applications is presented, with a focus on GUI bugs [9]. First, a bug mining study is conducted to understand the nature and frequency of bugs affecting Android applications.

The contributions of this paper include the following.

1. A bug study and categorization of Android-specific bugs that shows an important number of Android bugs manifest themselves in a unique way that is different from traditional, e.g., desktop/server application bugs.
2. An effective approach for detecting Android GUI bugs, based on a combination of test case and event generation with run-time monitoring and log file analysis.

For each class of bugs, the authors we were able to rediscover bugs already reported as well as new bugs that have not been reported yet.

5.4.1 Activity Bugs

Activity bugs stem from incorrect implementation of the Activity class, e.g., one activity might be created or destroyed in the wrong way so that it will make the application crash. In general, activity bugs occur either because developers are not sufficiently familiar with the activity- and event-based application model in Android, or because the implementation fails to obey the activity state machine.

The authors were able to detect all 8 activity errors that have already been reported , as well as 3 new activity errors in the apps Andoku and Connectbot.

5.4.2 Event Bugs

Android applications should be prepared to receive events, and react to events, in any state of an activity. If developers fail to provide proper implementations of event handlers associated with certain states, the application can either enter an incorrect state or crash outright

The technique has detected 24 event errors (18 existing bugs and 6 new bugs).

5.4.3 Type Errors

The authors successfully detected all old type errors , but not find any new ones.

5.5 Targeted Event Sequence Generation

Keywords:

- **Symbolic execution**: Symbolic analysis is applied to each event handler to produce an event handler summary characterizing its behavior. This summary includes necessary data and control flow information about every execution path in the event handler code.
- **Concolic execution**: Concolic testing enables automatic and systematic testing of programs, avoids redundant test cases and does not generate false warnings. Experiments on real-world software show that concolic testing can be used to effectively catch generic errors such as assertion violations, memory leaks, uncaught exceptions, and segmentation faults.
- **Anchor events**: They are a small set of events, that are responsible for setting the necessary program state for a target to be executed.Here, the target generally requires execution of a series of event handlers that mutate the program state executing the event handler that contains the target.
- **Connector Events**: They are a disjoint set of events used only for connecting the initial state, the anchor events, and the target. These connector events do not affect the program state used at any anchor event or at the target.

Motivational Statements:

- Anand et al. approached automated testing for Android applications using concolic execution. However, their approach explores the application starting from the its entry point, not aiming for particular targets
- Monkey is a popular random testing tool for Android that has been shown to be effective for bug finding .Other tools like A2T2, AndroidRipper , iCrawler, and

EXSYST enhance random testing by using the application GUI to guide the testing. These light-weight techniques can be a good starting point for automated testing. However, as they have a black-box view on the application code, they are generally unable to reach the challenging targets

- Tools such as Dynodroid employ feedback-directed automated testing, which is based on random testing but prioritizing using information gathered during the testing. Such techniques can often obtain good coverage with fewer test inputs that involve symbolic execution.However, they are not suitable for the more challenging targets.

The authors [6] proposed a two-phase technique for automatically finding event sequences that reach a given target line in the application code. The first phase performs concolic execution to build summaries of the individual event handlers of the application. The second phase builds event sequences backward from the target, using the summaries together with a UI model of the application.Thus approach is inspired by the work of Ma et al.[11] who consider the line reachability problem for C programs.

The contributions of this paper include the following.

1. The experiments on a collection of open source Android applications showed that this technique can successfully produce event sequences that reach challenging targets.
2. An important part of this approach is how concolic execution is applied to individual event handlers and the resulting summaries are composed for reasoning about event sequences.
3. The experimental results show that the approach can successfully cover challenging targets that are beyond the reach of random testing and conventional model-based test sequence generation sequences.

Targeted generation of application inputs can be useful not only for maximizing coverage in automated testing but also for reproduction of reported errors and evolution of test suites.

The results can be summarized as follows:

1. Challenging targets in real-world Android applications: The authors considered only the targets where all the baseline tools fail, the capability of the proposed algorithm to produce test cases for challenging targets is analyzed. Using this, 7 of the 16 targets of interest in Andriod app TippyTipper are reached.
2. The use of anchors and connectors and the effect on the the ability to reach the targets: Of all the generated test cases excluding the outlier, 53 perc of the events are connector events.The idea of pruning potential anchors by checking consistency of the symbolic states and the path conditions reduces the search space by eliminating 38 p.c-71 p.c of the anchors
3. Prioritization heuristics and the effect on the ability to reach the targets: For the half of the branches reached out of the full search space,enabling prioritization helped the total running time for the s equence generation to increase from 45 seconds to 2.5 hours.Thus, the authors showed that the prioritization heuristics have a considerable impact on the ability to reach targets within reasonable time.

5.6 Dynodroid: An Input Generation technique

Keywords:

- **Observe-Select-Execute cycle**: The tool presented runs on this principle where it first observes which events are relevant to the app in the current state, then selects one of those events and executes the selected event to yield a new state iin which the process is iterated.

Motivational Statements:

- Fuzz-Testing generates a sequence of random UI events.Fuzz testing is a black-box approach, it is easy to implement robustly, it is fully automatic, and it can efficiently generate a large number of simple inputs.Drawback: Monkey only generates UI events, not system related events. It would be challenging to randomly generate system events given the large space of possible such events and highly structured meta-data.
- Systematic-Testing automatically partitions the domain of inputs such that each partition corresponds to a unique program behavior. Thus, it avoids generating redundant inputs and can generate highly specific inputs.Drawback: In this approach,it is difficult to scale due to the notorious path explosion problem. Moreover, symbolic execution is not black-box and requires heavily instrumenting the app in addition to the framework.
- Model-based testing harnesses human and framework knowledge to abstract the input space of a programâĂŹs GUI, and thus reduce redundancy and im- prove efficiency program behavior. Thus, it avoids generating redundant inputs and can generate highly specific inputs.Drawback: In this approach, the tool predominantly target UI events as opposed to system events.

Dynodroid [4] views an app as an event-driven program that interacts with its environment by means of a sequence of events through the Android framework. By instrumenting the framework once and for all, Dynodroid monitors the reaction of an app upon each event in a lightweight manner, using it to guide the generation of the next event to the app.

Also, it also allows interleaving events from machines, which are better at generating a large number of simple inputs, with events from humans, who are better at providing intelligent inputs.The authors [4] evaluated Dynodroid on 50 open-source Android apps, and compared it with two prevalent approaches: users manually exercising apps, and Monkey[15]. Dynodroid, humans, and Monkey covered 55 p.c, 60 p.c., and 53 p.c, respectively, of each appâĂŹs Java source code on average.

The contributions of this paper include the following.

1. An effective system based on a novel âĂIJobserve-select-executeâĂİ principle that generates a sequence of relevant events. To adequately exercise app functionality, it generates both UI events and system events, and seamlessly combines events from human and machine.

2. Execution of Observe, select, and execute system events for Android in a mobile device emulator without modifying the app. The insight is to tailor these tasks to the vast common framework against which all apps are written and from which they primarily derive their functionality

3. Empirical evaluation of the system, comparing it to the prevalent approaches of manual testing and automated fuzzing, using metrics including code coverage and number of events, for diverse Android apps.

The authors reported Empirical evaluation of Dynodroid on real-world Android apps, and compared it to two state-of- the-art approaches for testing such apps: manual testing and automated fuzzing. The results of the evaluation are given in the repository [1].

5.6.1 App Source Code Coverage

Dynodroid runs 5X slower than Monkey primarily due to performance issues. On the plus side, as shown, Dynodroid achieves peak code coverage much faster than Monkey[15], requiring far fewer than even the threshold of 2,000 events the authors generated.Monkey triggers a large variety of UI events but no system events.The kinds of UI events that Monkey can generate is strictly a superset of those that Dynodroid can generate

Both Dynodroid and Human cover 4-91 p.c of code per app, for an average of 51p.c. Dynodroid exclusively covers 0-26p.c of code, for an average of 4p.c, and Human exclusively covers 0-43p.c of code, for an average of 7p.c. In terms of the total code covered for each app, human easily outperforms Dynodroid, achieving higher coverage for 34 of the 50 apps.

Dynodroid could be used to automate most of the testing effort of Human, as measured by what is the automation degree, measured as the ratio of coverage achieved by the intersection of Dynodroid and Human, to the total coverage achieved by Human. This ratio varies from 8p.c to 100 p.c across our 50 apps, with mean 83p.c and standard deviation 21p.c. These observations justify DynodroidâĂŹs vision of synergistically combine human and machine.

5.6.2 Bugs Found in Apps

To demonstrate its robustness, Dynodroid was run on the 1,000 most popular free apps from Google Play.We also found that Dynodroid exposed several bugs in both the 50 open-source apps. They are cross-checked for only FATAL EXCEPTION in the store, as this exception is the most severe and causes the app to be forcefully terminated.

5.7 Evodriod: An Evolutionary approach to testing

Keywords:

- **Evolutionary Testing**: Evolutionary testing is a form of search-based testing, where an individual corresponds to a test case, and a population comprised of many individuals is evolved according to certain heuristics to maximize the code coverage.

- **Application development framework**: It allows the programmers to extend the base functionality of the platform using a well-defined API. It also also provides a container to manage the life-cycle of components comprising an app and facilitates the communication among them.

Motivational Statements:

- K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution:Showed Evolutionary testing to be effective when sequences of method invocation are important for obtaining high code coverage. The work is limited to unit level. However, when applied at the system level, it cannot effectively promote the genetic makeup of good individuals in the search.

- L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service:EXSYST approach uses evolutionary algorithm in conjunction with GUI crawling techniques. This represents test suites as individuals and tests as genes.It generates tests that correspond to random walks on the GUI model. However, using EXSYST, the overall coverage is no better than the initial population

Mahmood et.al [19] presented Evodriod - an evolutionary approach to testing of apps. All the prior research has primarily focused on either unit or GUI testing of Android apps, but not their end-to-end system testing in a systematic manner.EvoDroid overcomes a key shortcoming of using evolutionary techniques for system testing, i.e., the inability to pass on genetic makeup of good individuals in the search. To that end, EvoDroid combines two novel techniques:

1. an Android-specific program analysis technique that identifies the segments of the code amenable to be searched independently, and

2. an evolutionary algorithm that given information of such segments performs a step-wise search for test cases reaching deep into the code

The authors ran tests on 10 open source apps to evaluate the line coverage between EvoDroid, Monkey, and Dynodroid.On average EvoDroid achieves 47 pc and 27 pc higher coverage than Monkey and Dynodroid, respectively. The results are discussed below:

5.7.1 Impact of Complexity

The analysis was carried by the complexity of apps. As the complexity class of apps increases, the coverage for Monkey and Dynodroid drops significantly. Since EvoDroid logically divides an app into segments, the complexity stays relatively the same, i.e., it is not compounded per segment. In all experiments, EvoDroid achieves over 98 pc code coverage. The cases where 100 pc coverage is not reached is due to EvoDroid abandoning the search when reaching the maximum number of allowable generations.

Once Monkey traverses a path, it does not backtrack or use any other systematic way to test the app. Therefore, MonkeyâĂŹs test coverage is shallow.As the depth of the complexity of apps increases, the time to execute EvoDroid increases. The results also show that Monkey runs fairly quickly, while Dynodroid takes longer.

5.7.2 Impact of Constraints

Input constraint satisfaction is a known weakness of search based testing techniques. When the probability of constraint satisfiability for an app decreases, the line coverage drops significantly for EvoDroid. Android Monkey coverage stays the same as it takes the one path with no constraints and does not backtrack. The coverage for Dynodroid drops also, but remains better than Monkey, as it restarts from the beginning several times during execution.

The results demonstrate that EvoDroid performs poorly in cases where the apps are highly constrained . Fully addressing this limitation requires an effective approach for solving the constraints, such as symbolic execution,as discussed earlier in [10][18].

5.7.3 Impact of Sequences

Given the event driven nature of Android apps, there are situations when certain sequences of events must precede others or certain number of events must occur to execute a part of the code. EvoDroid for these types of situations by generating apps from higher complexity class with ordered sequence lengths ranging from 1 to 5. Sequences of events with these lengths would have to be satisfied, per segment in all paths, in order to proceed with the search.

While EvoDroidâĂŹs coverage decreases, it does so at a much slower pace than Monkey or Dynodroid, EvoDroid is effective in generating system tests for Android apps involving complex sequence of events. This is indeed one of the strengths of EvoDroid that is quite important for Android apps as they are innately event driven.

The key contribution of the authors work are:

1. an automated technique to generate abstract models of the appâĂŹs behavior to support automated testing
2. a segmented evolutionary testing technique that preserves and promotes the genetic makeup of individuals in the search process
3. a scalable system-wide testing framework that can be executed in parallel on the cloud.

6. AREAS OF IMPROVEMENT

Few areas of improvement in the research was identified while reviewing associated and cited literature.

1. Scaling to apps with native code : Most of the algorithms proposed above does poorly on the codes written using third party libraries. This could be one the area where authors could improve.
2. Input conditions : Reasoning about the inputs is something the authors could better deal with,especially in Evodriod.This is a known limitation of search based algorithms in general practice, such as evolutionary testing.
3. OS Support : Most techniques support certain versions of Android. This may hinder its adoption for a fast-evolving platform like Android. For instance, this problem could be solved by the fact that Dynodroid instruments the SDK at the source-code level, and a patch could be created to use it in other Andriod versions.
4. Non-determinism : in programs is problematic for any dynamic analysis. One simple way to alleviate nondeterminism would be to treat all asynchronous operations synchronously in all the input generation techniques
5. In App Communication : Dynodroid restricts apps from communicating with other apps and reverts to the app under test upon observing such communication. However, many Android apps use other apps for shared functionality(ex.browsing). The authors could have addresses this problem since there are lot of dependencies.
6. UI input requirement : Limitation of model such as targeted sequence generation is that it requires UI models as input. This can be corrected by integrating existing frameworks for automatic UI model construction. Such a remedy would enable a larger scale experiment in which Android applications are automatically analyzed and tested.
7. Reaching other targets : Collider, the tool mentioned in [6] produces event sequences for challenging/unreachable targets. However, part of the remaining targets can also be reached using the prototype, provided that the symbolic constraint solver component is included with better support.
8. Reduce space via Abstraction : In ESESs [14],the authors has demonstrated some success with reducing the space via abstractions for instance using modal dialogues. Additional abstractions created in such framework will help to further reduce the space. This could potentially save a lot of space.
9. A metric to evaluate clustering : In the clustering of nodes [7] is the key step in abstraction activity. As observed, there was a difference in output using the two proposed criteria. It would be an helpful to have a metric like accuracy or precision after the tester made the grouping after evaluating the criteria.

7. CONCLUSIONS AND FUTURE WORK

The above Results have summarized different techniques ranging from input sequence generation, to GUI Ripping of the application. While the input sequence generation has led to quicker detection of bug or minimized the time to taken to find the first bug in code, Evodriod has shown to be significantly better than existing tools and techniques for automated testing of Android apps.In the worst case scenario it can degrade quite a bit due to its inability to systematically reason about input conditions.

However, with the combination of techniques such as symbolic execution [16], the above problem could be reduced.

To conclude, we suggest that search based testing is the way ahead given the ease of implementation and the scope of the coverage. It is quite flexible and there is quite a good amount of evidence which suggests so.

8. ACKNOWLEDGMENTS

We thank Prof.Menzies for giving us the opportunity to explore the content in the Automated Software Engineering. We have gained sufficient knowledge in this area, and

more importantly the procedure to go about the literature review. Also, we extent our gratefulness to NC State University Libraries which helped us find the relevant literature and citations with minimal effort.

9. REFERENCES

[1] dynodroid. *http://dynodroid.gatech.edu/study.*

[2] Guitar: A model-based system for automated gui testing. *http://guitar.sourceforge.net/.*

[3] Hierarchy viewer. *http://developer.android.com/tools/help/hierarchy-viewer.html.*

[4] R. T. A. Machiry and M. Naik. Dynodroid: An input generation system for android apps. In *2013 9th Joint Meeting on Foundations of Software Engineering*, page 224âĂŞ234, 2013.

[5] M. P. A. Memon and M. Sofa. Automated test oracles for guis. In *roceedings of ACM Conf. on Foundations of Software Engineering*, 2000.

[6] M. R. P. C. S. Jensen and A. MÃÿller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[7] A. F. D. Amalfitano and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *oftware Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference*, 2011.

[8] P. T. S. D. C. D. Amalfitano, A. R. Fasolino and A. M. Memon. Using gui ripping for automated testing of android applications.

[9] C. Hu and I. Neamti. Automating gui testing for android applications. In *Proceedings of 6th IEEE/ACM Workshop on Automation of Software*, 2011.

[10] K. M. J. Jeon and J. F. Symdroid. Symbolic execution for dalvik bytecode, 2012.

[11] J. S. F. Kin-Keung Ma, Yit Phang Khoo and M. Hicks. Directed symbolic execution. In *Proc. 18th International Static Analysis Symposium*, 2011.

[12] J. King. Symbolic execution and program testin. In *Symbolic execution and program testing*, 1976.

[13] A. Memon and M. Sofa. Regression testing of guis. In *ACM Conf. on Foundations of Software Engineering (FSE*, 2003.

[14] A. M. Memon. An event-flow model of gui-based applications for testing. In *Softw. Test. Verif. Reliab*, page 17:137âĂŞ157, 2006.

[15] A. monkey. http://developer.android.com/guide/developing/tools/monkey.html.

[16] C. S. P. Ì. Ì. N. E. N. Mirzaei, S. Malek, R. Mahmood., and M. Hicks. Testing android apps through symbolic execution. In *. SIGSOFT Softw. Eng. Notes, 37(6):1âĂŞ5, Nov. .*, 2012.

[17] N. K. P. Godefroid and K. Sen. Dart: Directed automated random testing. In *Proceedings of ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[18] T. K. N. M. S. M. R. Mahmood, N. Esfahani and A. Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Proceedings of 7th IEEE/ACM Workshop on Automation of Software Test (AST)*, 2012.

[19] N. M. Riyadh Mahmood and S. Malek. Evodroid: segmented evolutionary testing of android apps. In *Proceeding FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.

[20] M. J. H. S. Anand, M. Naik and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59:1âĂŞ59:11, 2012.

[21] G. N. W. Choi and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications*, 2013.

[22] L. White and H. Almezen. Generating test cases for gui responsibilities using complete interaction sequences. In *Proceedings of 11th IEEE Intl. Symp. on Software Reliability Engineering (ISSRE), 2000*, 2000.

[23] M. C. X. Yuan and A. Memon. Generating event sequence-based test cases using gui runtime state feedback. In *Trans. on Soft. Engr.*, 2010.

[24] X. L. Zhifang Liu, Xiaopeng Gao. Adaptive random testing of mobile application*. In *2nd International Conference on Computer Engineering and Technology*, pages V2–301, 2010.