Automated Testing of Android Applications

Arjun Augustine
North Carolina State University
Raleigh, USA
aaugust4@ncsu.edu

Devan Harikumar
North Carolina State University
Raleigh, USA
dhariku@ncsu.edu

Jordy Jose
North Carolina State University
Raleigh, USA
jjose@ncsu.edu

ABSTRACT

The smartphone market is growing day by day. According to Google there are around 1.4 billion active Android Devices world wide and the number of Android Applications available in the Google Play Store is around 2.2 million. With so many applications available, it is important to optimize their development. It is a well-known fact that early detection of Software Bugs reduces the cost of Software Development. Hence, development of novel tools which can detect relevant Software Bugs in a relatively new platform like Android during the development phase itself is critical. This paper summarizes different tools and approaches that evolved in the last decade for Automated Testing of Android Applications.

Keywords

GUI Ripping; Rich Internet Application; Fuzz Testing

1. INTRODUCTION

Smartphones are now ubiquitous even though they are new systems whose security and automated testing infrastructure is largely underdeveloped. As of 2011, the android share of the smartphone market was over 52.5% which was double what it was in 2010. The android market exceeded 10 billion app downloads with a growth rate of 1 billion app downloads per month. The features of Android devices and the complexity of their software continue to grow rapidly.

A few problems ever since the boom of the Android smartphone industry from a software engineering perspective are defects related to:

Improper management of limited resources, like memory, that leads to slowdowns, crashes, and negative user experience, GUI oriented application construction paradigm of android and its novelty, which means many existing android software correctness issues fall outside the scope of traditional verification techniques. Lack of cost-effective approaches for development and suitable techniques and tools for testing Primitive testing technology, and device fragmentation, which leads to heavy reliance on manual testing and increased test effort due to the number of devices that must be considered. Android framework that tries to abstract a lot of intra-process communication, thread management and synchronization within itself during run-time, which leaves the developer opaque to such aspects giving rise to a lot of concurrency bugs unless the delveloper spends considerable time on understanding the semantics of Android. This paper looks at how automated testing of Android applications has evolved over time.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. RELATED WORKS

2.1 Using GUI Ripping for Automated Testing of Android Applications

GUI Ripping is a dynamic process in which the software's GUI is automatically traversed by opening all its windows and extracting all their widgets (GUI objects), properties, and values. The extracted information is then verified by the test designer and used to automatically generate test cases. Certain GUI exploration criterion decides whether GUI exploration using a particular input is to be continued or not. Certain metrics may help in making this decision - ensure the depth of resultant GUI tree is less than a specified maximum, or if the resultant state is equivalent to an already evaluated state. The authors implemented the AndroidRipper using the Robotium Framework and by the Android Instrumentation class. and used metrics like Defect Detection Effectiveness, Code Coverage Metric and Time (Resource) spent to arrive at a conclusion about how effective the automated test cases generated are, in detecting bugs in an application. The results (running on Wordpress - an opensource android blogging software with broad user community with an issue tracking system) were compared against the efficacy of Monkey - a non commercial random stress and crash testing tool for Android GUIs. It was chosen because that was the only non-commercial automated android testing tool "to the best of author's knowledge". The Monkey tool generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner. They found that whereas the Monkey executes for 4.46 hours firing 45,000 events with the default value of event type statistical distribution finding 3 crashes reaching 25.27% LOC coverage, the ripping tool covered about more than 35% of code in about the same amount

of time to detect multiple undocumented bugs in the WordPress application. However, the paper lacks in comparing the performance of the tool against additional tools other than the Monkey, while running on additional applications other than the Wordpress app that they used.

2.2 Automating GUI testing for Android applications

The study aims to automate android testing so as to detect more bugs (especially GUI bugs) and increase the quality of developed software. First, the authors conduct a bug collection and categorization on 10 popular open source Android applications. They selected Android-specific bugs, i.e., those due to the activity and event-based nature of Android applications. To detect and fix these categories of bugs, they employed test and event generators to construct test cases and sequences of events. These were fed to the application, and a log file analysis was done to detect potential bugs, which were compared to those already reported by users and documented on the app website. The software detected most bugs reported, and found new bugs which had never been reported. The apps were chosen based on popularity, lifetime, detailed bug history, open-source source code, and high bug category coverage. The categories that the software classified bugs into were activity bugs, event bugs and type errors. More classifications like unhandled exceptions, API errors, IO errors and concurrency errors are listed as future work. GUI Testing is a related work proposed by Kervinen et. al.: They created a model that tests mobile applications running on symbian platform. Their Guitar framework (GUI testing framework) for java and windows applications, generate test cases automatically using a structural event generation graph. Their approaches however, target Java desktop applications, which are quite different from the Android mobile environment. Maji et. al.'s Android Bug Studies did failure characterization study that showed that defect density tends to be lowest in the OS, higher in the middleware, and highest in core applications. The technique demonstrates considerable improvement over Monkey tool, but does not involve any associated data like code coverage, that would help identify the shortcomings and better the results.

2.3 Semantically Rich Application-Centric Security in Android

The existing Android operating system needs to be augmented with a framework to meet the security requirements of android applications. Applications statically identify the permissions that govern the rights to their data and interfaces at installation time. This means that the application/developer has limited ability thereafter to govern to whom those rights are given or how they are later exercised. Android framework defines some Application (Security) Policies that takes care of security in the system. They can govern a myriad of areas like what applications can be installed, which other applications or user can have access to what data inside another app, to whom to provide access to ones own interface and how to continue monitor fair and secure usage of the interface. The authors define a SAINT framework to let applications assert and control the security decisions on the platform. Secure Application INTeraction (Saint) framework extends the existing Android security architecture with policies for applications that address some key application requirements like:

- Control to whom permissions to use its interface can be granted
- Control how its interfaces can be used by applications that were permitted to use them, and
- Determine at run-time, what other interfaces can they use.

Policy enforcements by SAINT framework include:

Install-Time: An application declaring permission P can be installed only if the policy for acquiring P is satisfied. Run-Time: Interactions between a caller and a callee app is allowed only if policies supplied by both apps are satisfied. Administrative: An administrative policy dictates how policy itself can be changed. Operational: Policies that detect when Saint renders an application inefficient, to prevent Saint hampering utility. Saint was not compared to any other existing system since the systems for run-time validation of Security Permissions were not very developed. Frameworks which validate permissions during install time are:

Kirin - enforces that the permissions requested by applications are consistent with System policies. Open Mobile Terminal Platform - determines an application's access rights based on its origin. Symbian framework prevents unsigned applications from accessing 'protected' interfaces. MIDP 2.0 Security Framework relies on the Mobile Information Device Profile implementor in giving access. Since there are no parallels to compare the Framework to, the paper fails to depict the impact of this new Framework. The paper does not mention any experiments conducted on this new framework.

2.4 Systematic Testing for Resource Leaks in Android Applications

The idea behind this paper is that leaks in Android applications often follow a small number of behavioral patterns, which makes it possible to perform systematic, targeted, and effective generation of test cases to expose leaks, that arises from improper management of resources, and can lead to slowdowns, crashes, and negative user experience. The authors try to use Neutral Cycles (sequences of GUI events that should have a âĂIJneutralâĂİ effect and should not lead to increases in resource usage. Such sequences correspond to certain cycles in the GUI model) to detect leaks. Through multiple traversals of a neutral cycle (e.g., rotating the screen multiple times; repeated switching between apps; repeatedly opening and closing a file), several common leak patterns in Android applications can be exposed. GUI models are directed graphs, with one node per activity, and with edges representing transitions triggered by GUI events. In addition to traditional events, the model should capture Android-specific events like pressing the home button. The following resources are monitored while execution:

Java heap memory: This is the memory space used to store Java objects. There is a garbage collector, so there can be leaks only when unused objects are unnecessarily referenced. Native memory: This memory space is used by native code. It requires explicit memory management by the developers as in programs written in non-garbage-collected languages such as C/C++, and thus could suffer from all well-known memory-related defects in those languages. Binders: Binders provide an efficient inter-process communication mechanism in Android. Usage of binders

requires creation of global JNI references, and unnecessarily keeping these references could lead to leaking other large Java objects. Threads: A sustained growth in the number of active threads in an application is an indication of software defects. The authors used GUI Ripper to obtain models of potential target GUI models. They evaluated the proposed testing approach on eight open-source Android applications. The test cases were generated with their LEAKDROID tool, and all failing test cases and identified the underlying defects were debugged and classified Memory, Thread, Binder leaks and Unique defects. It is impossible to ascertain how many leak defects actually exist in the studied applications, but the number of leaks and defects detected in popular Android applications indicate the requirement for detailed model based testers to avoid common bugs in Android.

Related work include model-based GUI testing, where, given a GUI model, test cases can be generated based on various coverage criteria. In these approaches the focus is typically on functional correctness and the coverage criteria reflect this in contrast to testing non-functional properties in order to target common leak patterns. Random testing: For example, Hu and Neamtiu use the Monkey tool to randomly insert GUI events into a running Android application, and then analyze the execution log to detect faults. Random testing is highly unlikely to trigger the repeated behavior needed to observe sustained growth in resource usage. Testing and static checking for Android: Prior work has considered the use of concolic execution to generate sequences of events for testing of Android applications, such as testing of exception-handling code when applications are accessing unreliable resources. As an alternative to testing, static checking can identify various defects including invalid thread accesses, energy-related defects, and security vulnerabilities.

Still, the approach can go a long way given additional coverage criteria, better diagnosis techniques and considering user driven sensor inputs are implemented.

2.5 Race Detection for Android Applications

The paper speaks about a motivating example of an android application that is as ubiquitous as a media player. Even a simple execution scenario such as downloading a media from the web and playing it, involves four threads running in two different processes. Moreover, much of the complex control flow and inter-thread communication in this example is managed by the Android runtime itself and is opaque to the developer. The developer must understand the semantics clearly to avoid concurrency bugs.

A data race occurs if there are two accesses to the same memory location, with at least one being a write, such that there is no happens-before ordering between them. Based on the concurrency semantics, the authors define a happens-before relation over operations in execution traces. Specifically, it induces an ordering between two asynchronous tasks running on the same thread if they use the same lock. This is a spurious ordering since locks cannot enforce an ordering among tasks running sequentially on the same thread. They overcome this difficulty by decomposing the relation into (1) a thread-local happens-before relation st which captures the ordering constraints between asynchronous tasks posted to the same thread and (2) an interthread happens-before relation mt which captures the ordering constraints among multiple threads.

DROID RACER is a race detection algorithm that provides a framework that generates UI events to systematically test an Android application. It runs unmodified binaries on an instrumented Dalvik VM and instrumented Android libraries. A run of the application produces an execution trace (a task in the execution scenario of an android application that can be divided into a sequence of more transparent and concurrency-relevant low level operations), which is analyzed offline for data races by computing the happens-before relation. The control flow between different procedures of an Android application is managed to a large extent by the Android runtime through callbacks. DROID RACER uses a model of the Android runtime environment to reduce false positives that would be reported otherwise. Further, DROID RACER assists in debugging the data races by classifying them based on criteria such as whether one involves multiple threads posting to the same thread or two co-enabled events executing in an interleaved manner.

DROID RACER was applied on 10 open-source applications and 5 proprietary applications from different categories like entertainment, communication, personalization, and social networking. These include popular, feature-rich applications like Twitter and Facebook with more than hundred million downloads each. Some of the applications exhibited complex concurrent behavior by spawning many threads, starting Service components, and triggering Broadcast Intents, even before DROID RACER triggered the first UI event on those applications. For such applications, we triggered sequences of 1âĂŞ3 events only. For each application, DROID RACER found tests which manifested one or more races. This shows that data races are prevalent in Android applications.

Related Work:

There are some tools that target specific types of concurrency bugs in Android. Dimmunix is a tool to detect and recover from deadlocks. There are tools that check Android applications against specific thread usage policies; e.g., AndroidâĂŹs StrictMode tool dynamically checks that the UI thread does not perform I/O or other time-consuming operations, Zhang et al. statically check that only the UI thread accesses UI objects. The concurrency model exposed by Android is different compared to the models explored in the literature in the context of race detection. There is of course a large body of work on static and dynamic race detection techniques for multi-threaded programs, e.g., based on locksets or happens-before relations or their effective combinations. However, these algorithms do not consider asynchronous calls, and either do not scale or produce many false positives, if asynchronous calls are simulated through additional threads. Recent work on race detection for client-side web applications considers the happens-before relation for single threaded event-driven programs and framework-specific rules to capture the semantics of browsers and JavaScript. However, their analysis is not immediately applicable to Android because there is additional interference through multi-threading. In the authors words, "Android is an expressive programming environment, and our formalization does not capture all its features." For example, the handling of inter-process communication (except IPCs relating to lifecycle events) has not been formalized. DROID RACER only generates UI events but not intents in the testing phase. Modeling and implementing these additional features are left for future work.

If there are multiple races belonging to the same category on the same memory location, DROID RACER reports any one of them randomly. But if DROID RACER is to be used as a tool for Concurrency Semantics, all race conditons have to be exhaustively listed. The manual inspection to distinguish between true and false positives. (1) For multi-threaded and cross-posted races,stall certain threads using breakpoints, giving others the opportunity to progress or to enforce a different ordering of asynchronous procedure calls. (2) For co-enabled races, change the order of triggering events. (3) For delayed races, alter delay associated with asynchronous posts. The paper does not talk about how these variations are achieved.

2.6 MobiGUITAR Automated Model - Based Testing of Mobile Apps

People adopt mobile platforms largely because of the apps they offer. Hence, quality assurance techniques like Software Testing is very important. Even though there are different techniques for testing mobile apps they are mostly stateless and never handled security. This is a big limitation because mobile apps are extremely state sensitive and have enhanced security. So there is a need for novel techniques.

MobiGUITAR - Model GUI Testing Framework models the states of the app's GUI. It also have a test adequacy criteria based on state machines. The model and the criteria are used to generate test-cases and automate the app testing. It uses Breadth-First Search (BFS), an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors. The first step involved is Ripping: This step obtains a state-machine model of the appâĂŹs GUI to be used for test generation. They create the model via an automated reverse engineering technique called GUI Ripping.

The next step is Generation: This step obtains test cases, each modeled as a sequence of GUI events. As can be imagined, the number of all possible event sequences that may be executed on any non-trivial appâĂŹs GUI is extremely large (in principle infinite because of loops). The test generation strategy needs to sample from this space. The authors develop a pair-wise edge coverage criterion. Conceptually, this means that all pairs of adjacent edges (events) need to be exercised together. To this end, they create pairs of all edges in our state machine that are adjacent to a node. And for each pair, they generate a test case that is a path in the state machine from the start state to the pair being covered.

The final step is Execution: The current implementation of the test generator outputs test cases in JUnit format. Such test cases are able to detect crashes in the app during its execution. A tester may enhance a test case by adding JUnit-like assert statements to check for functional errors. For thier Aardict app, 674 of the 678 test cases executed without any problems and covered 70

The authors automatically executed MobiGUITAR on some subject applications; in all, among the 7,711 test cases that were generated and executed, several of these test cases revealed bugs; in all, they detected 10 bugs. The test cases obtained from the model were able to find all the seven bugs already detected during the Ripping step, besides three new ones. Some bugs showed that Android applications may have incorrect behaviors due to a wrong management of the lifecycle of their Activities.

The authors compare MobiGUITAR to two tools available for Android testing: Monkey[5] and Dynodroid[6]. They selected Monkey, a random testing tool that comes with the Android Development Toolkit, because of its popularity in the Android developers community. Dynodroid was chosen because of the variety of event-based testing techniques it implements. They configured Monkey and Dynodroid to test the same four apps we tested by MobiGUITAR by the same number of events that were fired by the tool. They analyzed the exceptions arisen by the tools and the appsâĂŹ lines of code causing them. They were matched against the ones found by MobiGUITAR and that were associated with bugs. Not all were able to debug all the other exceptions because of poor debugging support provided by Monkey and Dynodroid. Not only were both tools able to find only three of the ten exceptions, they did not find any additional bugs either.

3. BASELINE RESULTS

Most of the papers we studied had baseline results to compare the work. In paper 1 - Using GUI Ripping for Automated Testing of Android Applications, GUI ripping for automated testing of Android Applications was compared against Monkey - a non-commercial random stress and crash testing tool for Android GUI's. While the GUI ripping method was able to record 8 crashes in 4.58 hours, Monkey could record only 3. Moreover, the GUI ripping method could cover 37.83% of the code in 4.58 hours, Monkey method could cover only 25.27% of the code in 4.46 hours. The software bugs discovered by GUI ripping tool were new and relevant. Thus GUI ripping is definitely a reliable tool for testing Android applications and outperforms the most popular Monkey tool.

In paper 2 - Automating GUI testing for Android applications, a GUI based technique is used for testing Android Applications. This technique tries to detect bugs classified into three broad categories: activity bugs, event bugs, and type errors. While testing an Android application this method detected all 8 previously reported activity bugs and 3 new ones. It also detected 24 event bugs out of which 6 were new ones. Moreover, the method could also detect all 4 of the existing type errors.

Paper 5 - Systematic Testing for Resource Leaks in Android Applications, proposes a novel and comprehensive approach for testing for resource leaks in Android software. The following results were observed for different Android Applications:

1. APV: The application crashes with a large native memory footprint due to incorrect implementation in native memory reclamation. 2. ConnectBot. SSH client ConnectBot has a defect related to leaking of event listener objects. 3. KeePassDroid. When the application is first launched, it displays a list of database files in FileSelectActivity for the user to choose. When a database file is selected, a query is launched to retrieve the information in the file, and the result can be accessed through a Cursor object. The Cursor is remembered in a container so that it can be synchronized with the activity. The Cursor object is automatically cleaned up when its managing activity is destroyed. However, when we keep the same instance of FileSelectActivity alive, and come back to the selection list to select database files repeatedly, multiple Cursor objects would be saved in

FileSelectActivity. 4. K9. In K9, a popular email client, a leak was discovered when rotating the screen after an email message is selected for display. Since it crashes after only a few repetitions of the ROTATE neutral cycle, this is an example of a leak that can be easily observed and thus cause negative user perception of the application. The paper propose a systematic and effective technique for testing of resource leaks in Android applications. Neutral cyclesâĂŤsequences of GUI events that should not lead to increases in resource usageâĂŤare used to define test coverage criteria.

Paper 6 - Race Detection for Android Applications, explains a race detection algorithm for Android Applications. This algorithm is implemented in a tool called DROID RACER. DROID RACER was applied on 10 open-source applications and 5 proprietary applications from different categories like entertainment, communication, personalization, and social networking. These include popular, feature-rich applications like Twitter and Facebook with more than hundred million downloads each. Some of the applications exhibited complex concurrent behavior by spawning many threads, starting Service components, and triggering Broadcast Intents, even before DROID RACER triggered the first UI event on those applications. For such applications, we triggered sequences of 1âĂŞ3 events only. For each application, DROID RACER found tests which manifested one or more races. This shows that data races are prevalent in Android applications.

Paper 7 - MobiGUITAR Automated Model-Based Testing of Mobile Apps, explains MobiGuitar, an automated model-based testing technique for mobile applications. Four apps were selected for testing MobiGUITAR - Aard Dictionary, Tomdroid, Book Catalogue and WordPress. All these apps are open source and are maintained by an active community of developers. MobiGuitar detected 10 bugs in total out of which one is related to Activity and three concurrency. The activity bug was because of the incorrect management of the life cycle of the activity. Few bugs were because of bad inputs. There were 2 bugs because of incorrect code reuse. It also detected bugs depended on programming mechanisms that arenâĂŹt traditional but are typical of Android apps.

In paper 8 - Sapienz: Multi-objective Automated Testing for Android Applications, we studied Sapienz tool. This tool uses uses multi-objective search-based testing to automatically explore and optimise test sequences, minimising length, while simultaneously maximising coverage and fault revelation. The top 1,000 Google play apps were tested using Sapienz. It found 558 unique crashes. The crashing behaviour has been verified on real Android devices (as well as Android emulators). 14 have been confirmed to be genuine, previously undetected, faults, 6 of which have already been confirmed as fixed by their developers. These results demonstrate that Sapienz is a practical tool for Android developers as well as for researchers. Sapienz significantly outperforms (with large effect size) both the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, in 7/10 experiments for coverage, 7/10 for fault detection and 10/10 for fault-revealing sequence length.

The baseline results shows a tremendous improvement in automated testing of Android Applications over the years. Some of the new tools can detect relevant software bugs at the very early stage of development which could substantially reduce the cost of development and improve the quality of the applications.

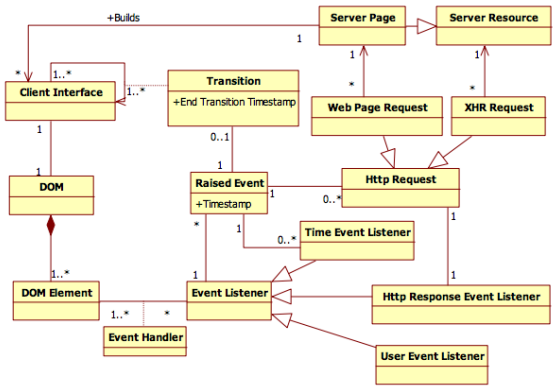


Figure 1: The conceptual model of a RIA client-side behaviour

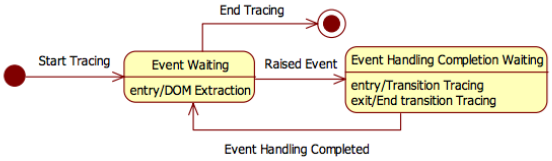


Figure 2: The Tracing Activity of the Extraction Step

4. INFORMATIVE VISUALISATIONS

As we all know a 'picture is worth a thousand words'. Visualizations are always helpful whether it is to represent complex information or to showcase baseline results. Almost all the papers we studied used different forms of visualizations like tables, graphs etc.

In paper 4 - Reverse Engineering Finite State Machines from Rich Internet Applications, we studied Reverse Engineering Finite State Machines from Rich Internet Applications. The paper uses a class diagram to represent the conceptual model of a RIA client-side behavior.

It uses an activity diagram to show the tracing activity of the extraction step.

It also uses component diagram to show the RE-RIA tool architecture.

Paper 7 - MobiGUITAR Automated Model-Based Testing of Mobile Apps, explains MobiGUITAR - a Automated Model-Based Testing of Mobile Apps. The paper uses abstract State-Machine for the app - Aard Dictionary, a dictionary and online Wikipedia reader. It shows different states and transitions between them making it easy to identify the different transitions which leads to the termination of the app.

Paper 8 - Sapienz: Multi-objective Automated Testing for Android Applications, explains Sapienz tool. The results on the 68 benchmark apps were shown in a very user-friendly and detailed manner. It includes the coverage, crashes and the length of the tests. It also includes progressive coverage on benchmark apps and pairwise comparison of found crashes. Additionally, performance comparison on 10 F-Droid subjects and the Vargha-Delaney effect size are also

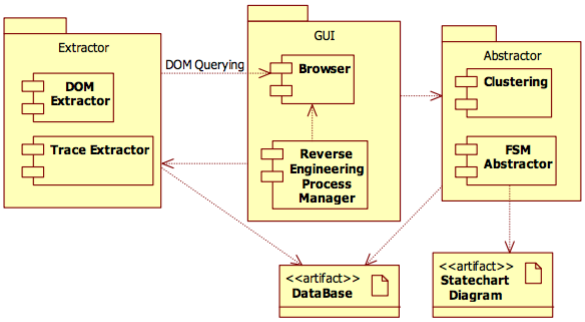


Figure 3: The RE-RIA tool architecture

tabulated.

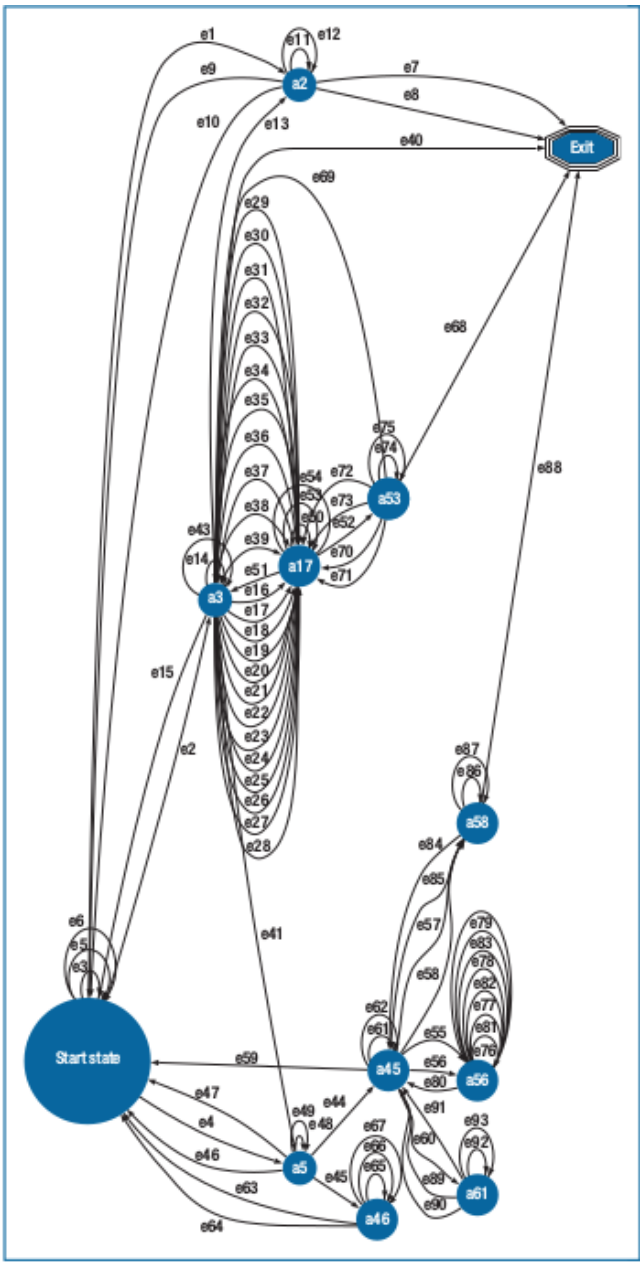
Visualizations makes it easier to represent complex information. It also makes it easier to grasp.

5. CONCLUSIONS

We have already seen some of the tools and approaches for Automated Testing of Android Applications, which emerged in the last one decade. One thing we observed from studying all those papers is that even though we made considerable progress there is still a lot of scope for improvement. For instance in paper 1 - Using GUI Ripping for Automated Testing of Android Applications, the Tool can be firther improved by augmenting the Tool's capabilities to find logical (semantic) Software bugs as well. In paper 6 - Race Detection for Android Applications, DROID RACER can be improved further to be used as a Tool for concurrency semantics.

All the papers we studied did not just focus on one way of testing Android Applications. They explored Automated Testing of Android Applications in different ways. They focussed on testing different things which would compliment each other. For instance, while some of them focussed on using GUI ripping to test an application, few others were focussed on Race detection, finding resource leaks, security breaches etc.

Software Testing is a very important phase of Software Development. The extend and quality of testing determines the quality of the Software and its production cost. It is less expensive to fix a Software bug earlier in the development stage than after it is deployed for production. There have been considerable improvement in the testing of Android Applications over the decade. However, there are areas which needs to be explored more.



6. ACKNOWLEDGMENTS

No content here

References

Author, Some (2005). *Book's title*. The City: Publisher.