TADS: Automating Device State to Android Test Suite Testing

Jonathan Sanders
Computer Science Department
University of Colorado
Colorado Springs, United States of America
jrsanders5411@gmail.com

Kristen Walcott
Computer Science Department
University of Colorado
Colorado Springs, United States of America
kwalcott@uccs.edu

***Abstract*—Android testing still has many areas to cover in order to make automated testing of Android applications as thorough as possible. One area of research that has yet to be investigated is automating testing of Android applications against changing device states. Device states can and do change during the usage of applications and do impact the usability of an app.**

This paper presents a novel approach to easily test multiple device states against an existing Android application's test suites. We built a tool that we called TADS (Test Application to Device State) that enables a developer to very easily run their Espresso test suites against multiple state changes in the device. In a small study, 100% of introduced errors were detected. The tool can easily be expanded for testing any number of states.

***Index Terms*—Automated testing, Automated mobile state testing, Automated mobile testing, Context Aware testing**

I. Introduction

Android applications have become a staple of many people's everyday mobile computing user experience. As of the first quarter of 2017 Android possessed 85% of the mobile phone market [3]. The need for complete and thorough testing capabilities of Android applications has never been greater. There is currently a deficit in an area of testing for mobile devices regarding the "state" of the mobile device while testing is taking place. Very often, particular states will cause a failure of a mobile application and currently there are not any good tools available to test different states to detect these types of failures.

The state of the device is the current settings of different modules within the mobile device such as if bluetooth is on or off or if airplane mode is on or off; other "states" include having different applications running that access common instrumentation or having hardware devices connected to the mobile device and running with an application such as a heart rate monitoring application running with a conected physical Electrocardiogram (EKG). Any configuration of these different instruments, applications, operating system configurations, or modules can be considered a device's state; hereto referred as *DS* for device state.

It is more expensive for a company to not test applications against DSs because the failures are often catastrophic. Companies have encountered these bugs after releasing an update for their application on the users' devices and the failure often causes a loss of credibility, loss of functionality that must then be quickly corrected, updated, and ultimately costs the company a loss of customers and money. Testing the different states of just an application has been enhanced with tools such as Espresso, Barista, and Robotium [1] but there are no automatic means of testing those application states against the device state. Since the state of an application can be easily saved and the state of a device can be easily changed programmatically, a solution that automatically tests application states with different device states is of great value.

We have developed a novel approach to very easily run existing Android test scripts against any set of device states. The solution we built is called "TADS" (Testing Application to Device State). The tool uses Espresso, an automated Android application testing tool, to test the Mobile application against multiple DSs. This especially allows us to test context aware applications. These are applications that are able to detect information about the device's physical environment using instrumentation and then process that information.

TADS has been able to run multiple tests against multiple DSs utilizing two Device State Changes ((DSCs). We focused on Wi-fi and Bluetooth device states in this work. The results show that 100% of our tests developed in TADS were able to detect the mutant error states that we introduced to our programs.

The main contributions of our paper are:

- Highlights and addresses the lack of Data State testing in mobile applications. (Section II)
- Description of TADS: a library for performing state testing of devices (Section III)
- Creation of TADS to automate testing for context aware devices (Section III)
- Discusion of the effectiveness and impacts of TADS (Section IV and VII)

II. Mobile Testing Automation

Mobile testing is a challenge due to the the large complexity in their inputs. These devices are context aware. We will first discuss what context aware applications are and then describe several tools that have been used to test code for mobile devices.

A. Context Aware Applications

A "context aware" application is an application that is able to detect information about the device's physical environment and use the information based on the context observed. This is very common in mobile and embedded environments. For instance, a context aware application can get GPS information and see that a user is at work. Then it can get location information from the network to see where in the building the user is, then get information from the calendar application about what is scheduled at that moment. Then the context aware application will determine that the user is in a board room in their office building during a scheduled meeting and puts the phone on silent automatically or even stops all calls and replies with a text message saying the user is unavailable.

DS testing is critical for these types of applications. If another application acccessing certain information collides with the context aware application's access of that data, the context aware application could fail. Currently, there is not a way to determine if that will happen using common automated testing tools, which is what TADS attempts to guard against by focusing on state.

B. Mobile Testing

Several tools have been recently developed to aid in mobile testing. These mainly focuson event tracking for later automation.

Espresso is a tool developed by Google that enables a user to easily record a series of events and then create an oracle so that the user can create an automated test case. An oracle is a way of knowing if the series of events executed as expected. The Espresso recorder autmatically generates a testing script that can be executed at any time to recreate the series of events and check the oracle to ensure the series of events executed correctly. The script then generates a pass or fail message that the user can see to know if any recent development work broke the existing software application [1].

Fazzini et al. then created Barista, which advances the way to record and execute Android testing from that of Espresso. Barista is an application that records user interactions with any Android application and automatically can generate oracles and then it records those interactions and oracles into an Espresso type script for later execution [5].

TADS can be easily integrated with Barista to enable the user to run Espresso scripts generated by Barista with different DSs. TADS is built out of the Microsoft Powershell application. TADS also utilizes the Android Device Bridge and Android Studio. Android Studio is Google's integrated development environment for developing Android applications. Android studio generates a project when an Android application is started and within that project is a testing project. When a developer makes an automated test those scripts are included in that test project. Then, when the application is installed on a mobile device with the testing project attached (which is the default behavior of Android Studio), the test scripts are included on the device and can be executed via the Android Device Bridge.

Fig. 1. TADS work flow (contents of a DSC)

Step #	Steps	Example
1	Set Device State (DS)	Bluetooth on
2	Run Espresso Script	testScriptFile.Java is executed
3	Report Results	Espresso Results
4	Set Device State	Bluetooth Off
5	Run Espresso Script	testScriptFile.Java is executed
6	Report Results	Espresso Results

Fig. 2. Public Interface

Function Name	Explanation
invoke-TADSExecuteAllDSCsWIthAllTestScripts()	Run all tests in suite and all DSCs
invoke-TADSExecuteAllDSCsWIthOneTestScript()	Run 1 test case in Android Espresso Suite and all DSCs
invoke-TADSExecuteOneDSCsWIthAllTestScripts()	Run all tests in suite and 1 DSC
invoke-TADSExecuteOneDSCsWIthAllTestScripts()	Run 1 test case in Android Espresso suite and one DSC

III. TADS Implementation

The main purpose of TADS is to simplify testing of different device states an Android application. There are preloaded "device state changes" (DSCs) that can be used with a premade Espresso Test Script that are chosen for testing. A DSC is a test case that starts with a device state then runs the test cases and then changes the Devices state, as shown in Figure 1. The DSC then runs the tests again with the new state set. Another example of a DSC is: airplane mode off, run Espresso test script(s), switch airplane mode to on, then re-run the Espresso test script(s).

TADS is developed in Microsoft Powershell. Powershell is an enhanced version of the MSDos program. Powershell includes the .NET runtime and libraries which enables a developer to leverege the tools that .NET provides. There is a Powershell Integrated Development Environment called Windows Powershell ISE. With ISE all of the results of executing an ADB command can be viewed and recorded if desired. Powershell also has the ability to organize code into modules and plain Powershell scripting files. The modules are suffixed with a file type of .psm1. The normal script files are suffixed with the file type .ps1. TADS is made up of many functions in several modules and Powershell files that execute ADB commands from a computer that the device is connected to.

The implementation allows a user to call a function to test the following as is shown in Figure 1 The user can run all of their test suites and all of the DSC's available, all tests and 1 DSC that is selected, 1 DSC and 1 test case in the suite, and all DSCs and 1 test case. The name of the application should be the whole app name starting with "com". The appropriate

global variables should be filled in prior to calling the testing function. The public interface in Figure 1 is what novice programmers should use.

All of the files are available to be seen and modified. New DSCs can be easily added using the idioms of the TADS solution in order to make very thorough test cases.

The TADS solution is built on Microsoft Powershell and written as Powershell scripts. Powershell was decided on to enforce DRY and SOLID principles as much as possible and to make running from the command line as easy as possible. DRY and SOLID principles are ways of writing code that is clean and easy to reuse.

The TADS solution is run by executing a powershell script. There are global variables that must be populated by the user to indicate several items. Figure 2. demonstrates those global variables. They are modified in the file called "Project Main.ps1" The easiest way to execute TADS testing is via the Powershell ISE.

Figure 2 is a break down of the functions to be called by your command line or in a script that a developer would write to test their application how they deem necessary. The function applicable to what the developer is testing will be used.

There is a readme.txt file that should be followed to get the testing environment set up for usage of the TADS solution and there are thorough instructions included in the readme.txt file. The TADS solution can be downloaded at: https://github.com/UCCS-CS5371-Fall2017/jsander7/tree/master/ProjectFiles

Furthermore, all functions in the entire solution are:

- Available to be consumed by the user
- Globally scoped to avoid the challenge of Powershell scoping issues
- Named according to an idiom of Powershell using the TADS prefix.

For instance with the function "invoke-TADSExecuteAllDSCsWIthAllTestScripts()", "invoke-" is the Powershell idiom (there is a list of these online) and then the prefix TADS is added so as to avoid collision with other global functions that are not a part of the TADS solution, this is also a Powershell idiom, and then a description of the function's purpose. This is so that a programmer can build their own Powershell script out of the existing functions and extend what has been made up to this point. As seen in Figure 3, within the TADS solution there are four main files. They are: ProjectMain.ps1, DSCs.psm1, TestSuiteCommands.psm1, and ADBCommands.psm1.

Fig. 3. Contents of Files

File	Contents
ProjectMain.ps1	Global Variables for user input: AppName, JunitRunner,TestName (for running one Espresso Script), DSCName (for running one DSC)
	Area to build script to run pertinent DSCs.
DSCs.psm1	Airplane Mode DSC code.
	Bluetooth DSC code.
	Execute all DSC code.
TestSuiteCommands.psm1	Public interface from Table 1.
	Code to generate needed global variables from user global variables.
	Code to run a specific DSC
ADBCommands.psm1	Set bluetooth on.
	Set bluetooth off.
	Set airplane mode on.
	Set airplane mode off.
	get airplane mode.

A. ProjectMain.ps1

ProjectMain.ps1 is where the developer will set into global variables the following:

1. The name of their application project application's Espresso script name
2. Type of Junitrunner
3. A DSC name if the developer wants to run one DSC only

The application project name will automatically be use to make the test suite name needed by the ADB shell to run the Espresso test suite. This file is also a convenient space for a developer to call DSC tests and to establish a test script using the TADS tool.

B. DSC.psm1

DSC.psm1 is where the actual DSC tests are written. The DSCs are individual functions. The DSC function will run on all scripts or just one script depending on what the developer decides to do. The readme has more precise information on the details of using TADS. The idiom used for these functions is PowershellPrefix-TADSDSCdescriptor.

C. TestSuiteCommands.psm1

This file is where the global variables filled out on the ProjectMain.ps1 file are used. These are also where the functions from Figure 1 reside. These functions are what actually execute the DSCs and there is a function in there for running a specific DSC that the other public interface functions consume.

D. ADBCommands.psm1

The ADBCommands file is where the state change commands are located that use ADB to change the states of the device. We have created easier to use functions than actually calling the ADB commands to change these items. Additioanlly, logging takes place in these functions which is critical to the usefulness of TADS. The idiom used with these functions is "TADSADBdescriptors".

IV. Evaluation of TADS

TADS has been able to run multiple tests against multiple DSs utilizing two DSCs. The DSCs made are for bluetooth and airplane mode. The bluetooh DSC turns bluetooth on, executes the test case generated, and then turns bluetooth off and executes the test case. Airplane mode does the same for airplane mode.

We utilized a recipe app for initial testing. We recorded the selection of a recipe and the assertion verified that recipe appeared. The second test involved selecting a recipe, ensuring the recipe appears, and then going back to the home screen of the app and ensuring the recipe list appears.

The recipe app was actually created for android wear as well. TADS easily handled testing this application which connects to an android wear device.

The results were passed tests. We did not attempt to create any mutation tests by modifying code because the actual test is generated by the android test recorder, Espresso.

We evaluated the affectiveness of the application by adding code to the application that caused an exception to be thrown if bluetooth was off. We did the same if wifi was turned off to create an exception so that the airplane mode tests would show that having one state passes and another state causes a failure of the application.

These were simple tests but it proved the idea that TADS demonstrates. 100% of the time these tests caught the generated errors. Additionally, TADS adds test cases to be run. Just two DSCs adds 4 test cases that are viable and valuable tests to be run.

V. Related Work

There are several tools developed that can capture various information of an application and store said information for later tests. These tools can be leveraged to create interesting app states instead of simple object states. For instance Paulovsky et al. [8] built a tool that automatically captures UI information as a user utilizes an application.

Many others, such as Fazzini et al [5] have made tools for recording tests that can be later executed in ways that are platform independent which is a nice utility that we will not be concerned with in this work. Others have made unit level state testing models such as MilaniFard el al. [7] in which the state of the application is tested by automatically building a model using a dynamic and static crawler and then running that created model against a verification algorithm. Their work levereges data modeling to find issues concerning device state whereas TADS utilizes real world scenarios executed on real world devices.

G. Bai et al [2] developed a way to use model testing to find security vulnerabilities. Choi et al have used machine learning to find the states of an application that are probably of value to test that have not been tested and make a model of those states for testing [4]. Their work could be applied by developers utilizing TADS to develop DSCs appropriate for their application type.

VI. Threats to Validity

One serious threat to this work is that the automated test scripts built in Espresso can be affected by state changes [5] which can cause a failure in the test due to the script's failure but not the application's. That failure will result in a false negative test outcome. This error will be revealed by the logging utility, and corrections can be made a test design time. These false negatives could cost several hours in developer time per failure.

The case study also used was limited. However, the applications used for testing are applicable for both mobile and Android Wear applications and more. The tool can easily be extended for monitoring and evaluating context-aware applications beyond these that are presented.

Mutation testing or user-based testing should also be applied. While the tool works well, our manually generated errors provide potential bias.

VII. Discussion and Future of TADS

One challenge when working with Andoid applications is that commands do not exist for ADB to change proprietary device instrumentation states. Instantiating different hardware devices from the command interface and easily implementing new DSCs will prove extremely beneficial in making robust state testing of Android applications using proprietary instrumentation such as a portable Electrocardiograms.

Barista [5] provides a better script generation tool than Espresso. Integrating TADS to run Barista could prove to be a valuable endeavor. Also, being able to inject DSCs into Barista generated testing scripts will enhance the ability of Barista to catch bugs introduced by device state changes.

We may also expand the instrumentation capabilities to include wearable devices such as Android watch or even google glass for state testing with apps running on those devices. Currently TADS runs those applications on the device without any issues, but we have not yet tested the capability of TADS on an android wear device.

Another approach that could prove extremely beneficial is to evaluate the affect of these states and state changes on the efficiency of an application. One could create a way of capturing

pertinent throughput data and run some form of an evaluative algorithm against that data thereby giving developers pertinent information on how to improve their application's efficiency by using some kind of brownout technique [6] or other viable solution when that device state is detected and known to cause lagging throughput.

Another interesting area of research would be to apply the work by Choi et al [4] in order to determine which DSCs to run in a TADS test suite. Their research uses an algorithm to evaluate an application's source code to determine what states should be tested.

The last future research we are considering at this time is to investigate how device state affects installs. Installations can be affected by the state of different items running on a device and building a test script that installs an application and running that installation on devices with different states may prove valuable; though, some research into whether or not industry struggles with installations being affected by state would be an interesting approach.

VIII. Conclusion

The testing of Andoid applications is a challenging task. It is much more difficult due to the many numbers of states that the device may be in. In this work, we have created a state based tool to automatically check states on an Android device for testing purposes. The tool works on the device itself and in emulation.

Two settings were manipulated in our experiment- WiFi and Bluetooth. The tests that were generated detected 100% of the generated errors.

In future work, we intend to modify TADS to use Barista and expand the analysis of the tool to wearable devices. We would also like to perform mutation analysis or fault analysis based on states for better defect detection and increase our test base.

References

[1] Top 10 mobile testing tools, Nov 2016.

[2] G. Bai, Q. Ye, Y. Wu, H. Merwe, J. Sun, Y. Liu, J. S. Dong, and W. Visser. Towards model checking android applications. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[3] M. N. K. Boulos, A. C. Brewer, C. Karimkhani, D. B. Buller, and R. P. Dellavalle. Mobile medical and health apps: state of the art, concerns, regulatory control and certification. *Online journal of public health informatics*, 5(3):229, 2014.

[4] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, Oct. 2013.

[5] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. Barista: A technique for recording, encoding, and running platform independent android tests. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 149–160, March 2017.

[6] C. Klein, M. Maggio, K.-E. Arzen, and F. Hernandez-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 700–711, New York, NY, USA, 2014. ACM.

[7] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 67–78, New York, NY, USA, 2014. ACM.

[8] F. Paulovsky, E. Pavese, and D. Garbervetsky. High-coverage testing of navigation models in android applications. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*, pages 52–58, May 2017.