Learning User Behavior for Mobile Test Suite Adequacy

Cody Kinneer
Allegheny College
Email: kinneerc@allegheny.edu

***Abstract*—Software development for mobile devices proceeds at a rapid pace. Software as a service, rapid development, and agile programming means that mobile applications are released and updated quickly. As a result, developers have less time to test their applications and cannot completely know the effects of a change. Existing test suite adequacy criteria are insufficient in this quickly changing environment.**

In this paper, we develop a new behavior based test suite adequacy criterion that adapts to user interactions with an application in the wild. We evaluate the time and space overhead of they system and perform an empirical study analyzing existing Android test suites according to our behavior driven criterion. Our analysis reveals that the two test suites focused testing on infrequently used contexts, achieving behavioral adequacy scores from 12 to 33 percent less than the probabilistic calling context coverage. This shows the potential for substantial improvement in the development of test suites for mobile applications.

I. Introduction

Developers frequently use test suites, a collection of test cases, to ensure that the component under test performs according to specification, or to ensure that accuracy does not change over time. A test suite's usefulness lies in its ability to detect problems. With the rise of software as a service and rapid development practices, test suites must be effective in detecting important problems quickly. However, since it is not known beforehand where a problem will occur, determining the adequacy of a test suite is a challenging problem.

The most common test suite adequacy criterion is structural coverage. These criteria, such as line or block coverage seek to maximize the amount of code exercised by a test. Since a bug needs to be executed to be exposed by a test, maximizing structural coverage is a reasonable strategy. However, this definition of test suite adequacy suffers from not taking into account the importance of the structure covered. Achieving complete test coverage in practice is most often wishful thinking, and structural adequacy fails to provide insight into what areas of the application are more important to test. Furthermore, a good test should resemble the conditions under which the application will actually be used, but structural techniques say nothing about the realism of a test.

Another approach to test adequacy is fault-finding. This consists of introducing faults into an application, and then determining which tests tend to find the greatest number of faults. Mutation testing is one such technique. However, in practice, mutation testing speaks to the ability of a test to find faults in a certain structure. It cannot tell us what structures are more important to search for faults in.

With the rise of new software engineering paradigms such as software as a service, agile, and rapid development, these criteria fail to keep up with the pace of software development. This is particularly true of Android applications. According to Android's website, there are 7 Android API's in use [1]. The rate of change of the Android OS itself is a testament to the rapid development of Android applications.

These adequacy measures could be improved by taking into account the way users interact with the application after it is deployed. A behavior driven adequacy criterion confers two benefits. Firstly, if the purpose of application is to be used by a user base, then more frequently utilized components are more important than those that are less frequently used. A problem that occurs in a more frequent use area will affect more users. Additionally, a test suite's similarity to observed user behavior favors tests that are more similar to the conditions that the application will be exposed to in the wild.

Previous work in model-based software testing applied Markov chain models to software testing [2], [3], [4]. These works discuss how a Markov chain used to model software usage could be useful for input generation, software specification, and statistical software testing. However, they do not address the issue of how the model should be generated, and do not focus on test suite adequacy.

In this paper, we present a new test suite adequacy criterion that takes into account learned user behavior in the wild. By collecting data from users actually interacting with an application, we learn a Markov chain that models user behavior. This model can be continuously updated to respond to changes in the users' interactions. We then determine a test suites adequacy by its similarity to the constructed user behavior model.

We seek to determine how well test suites for Android applications reflect real user behavior. We evaluate our technique in terms of time and space overhead for a collection of Android applications, and evaluate the test suite adequacy of the applications using our proposed criterion.

The contributions of this paper are therefore as follows,

- A new behavior driven test suite adequacy criterion (section III).
- An implementation of the criterion for Android applications (section III and section IV).
- An empirical study evaluating the overhead of the implementation (section IV).

- An evaluation of several Android applications' test suites using the new criterion (section IV).

II. Background

To calculate behavioral adequacy, a technique called probabilistic calling context [5] is used for profiling and a Markov chain is used for modeling.

Profiling

Calculating a behavioral criterion requires that an application's behaviors be profiled. A program's behavior can be thought of as the collection of its function calls, which makes profiling based on these calls a reasonable choice for modeling application behavior.

Probabilistic calling context (PCC) is a profiling strategy developed by Bond and McKinley [5]. PCC attempts to assign an integer to every unique stack state. This system is useful because it can be computed efficiently, only 3% overhead is reported in the literature. An example of a stack state that could be represented by PCC is shown in Figure 1. This figure shows an example stack state inspired by the K-9 Mail email application. First, MAIN is called, which is assigned a PCC value of zero. Then, MAIN calls CHECKEMAIL, (hereafter, we will use $\rightarrow$ to signify a function call). The next PCC is calculated from the last PCC and the name of the CHECKEMAIL function. Thus, the next PCC value is meant to represent the sequence MAIN $\rightarrow$ CHECKEMAIL. If CHECKEMAIL were to return to MAIN, then the PCC value would also return to zero. Instead however, CHECKEMAIL calls ITERATEACCTS, so the next PCC value is calcuated from the previous PCC value and the next function name. When a function is called, the next PCC value is given by

$$nextPCC = currentPCC * 3 + currentContext$$

where $currentContext$ is an integer that represents the current context, such as a hash value of the called function name.

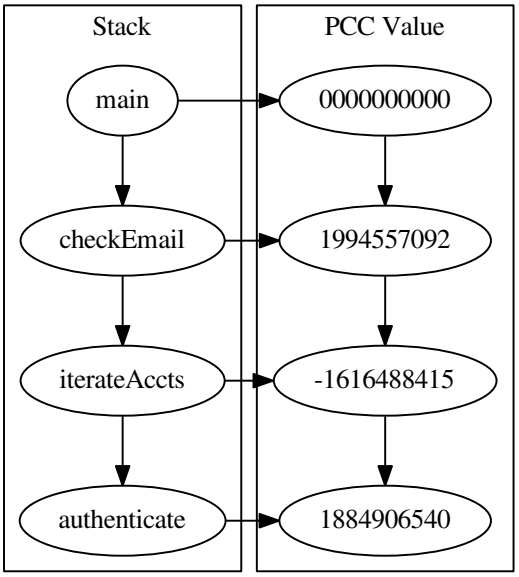

Fig. 1. PCC value updating as methods are added to the stack.

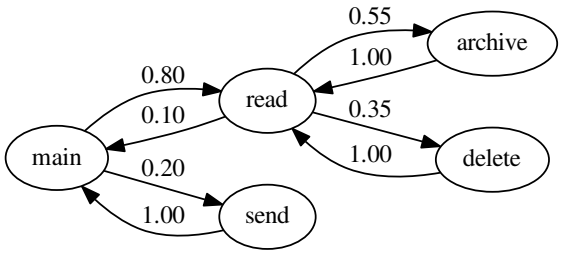

Fig. 2. An example of a Markov chain behavior model inspired by K-9 Mail.

Markov Chains

A Markov chain is a state based system where the next state depends only upon the current state [6]. An example is shown in Figure 2. The nodes in the graph represent states, and the edges represent the transition probabilities. Starting at the MAIN state, there is an 80% chance of transitioning to the READ state and a 20% chance of transitioning to the SEND state.

Markov chains have been used to model expected user behavior in model based testing [2], [4], [3]. However, these techniques generally do not learn models from user behavior, but reflect how the developer expects users to behave.

III. Behavior Driven Test Suite Adequacy

Using PCC and Markov chains, we present a technique for assessing a test suite's adequacy based on how users interact with the application being tested. Our technique for calculating behavioral adequacy is shown in Figure 3.

First, the application is instrumented to collect data for profiling user behavior. Because the application will be used while this information is collected, the overhead incurred by the user must be acceptably small. Additionally, the information gathered must be useful in modeling user behavior. PCC was chosen because it satisfies both of these requirements. A program's behavior can be thought of as the sequence of functions that it calls, which makes profiling based on function calls an appealing strategy for profiling behavior. Since PCC takes into account the functions on the stack, it provides more information than simply profiling based on function frequency, while still maintaining low overhead. The instrumentation calculates the current PCC value from the calling context, and records each transition between PCCs. The application is then released for use by the user base, and behavioral data is collected in the form of these PCC transitions.

The developer then runs the application's test suites on the instrumented application. The transitions between PCCs

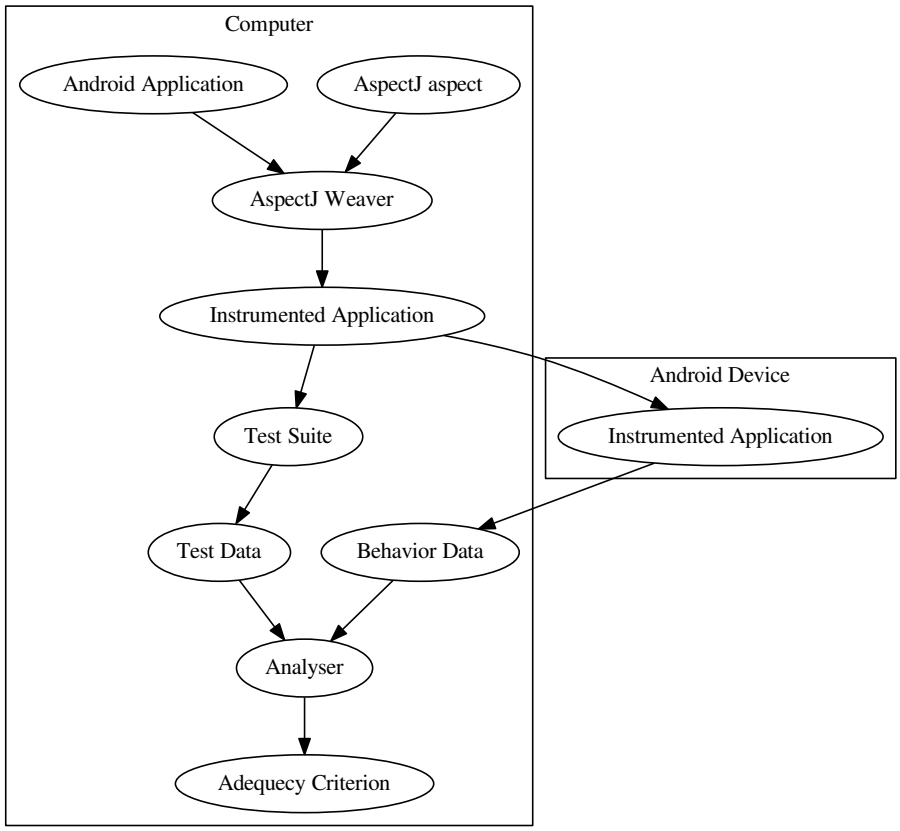

Fig. 3. Framework for a behavior driven test suite adequacy criterion.

observed during testing are recorded as well, giving test data that characterizes the behavior of the test suite in terms of the observed PCC transitions.

The user data is then aggregated and used to construct a Markov chain where the nodes are PCC values and the edges are the probability that a PCC value will transition into another PCC value.

For example, suppose the data in the table below was collected from users interacting with a simple email application. Caller PCC represents the current PCC value, and callee PCC represents the next PCC value. Rather than integer representations, the names of functions on the stack are given for explanatory purposes.

Caller PCC	Callee PCC	Frequency
MAIN	MAIN → READ	80
MAIN	MAIN → SEND	20
MAIN → SEND	MAIN	20
MAIN → READ	MAIN	8
MAIN → READ	MAIN → READ → ARCHIVE	44
MAIN → READ	MAIN → READ → DELETE	28
MAIN → READ → DELETE	MAIN → READ	28
MAIN → READ → ARCHIVE	MAIN → READ	44

This data would be converted to a Markov chain similar to what is shown in Figure 2. In the example, the nodes are function calls rather than PCC values for the sake of explanation. For example, the ARCHIVE node represents the context MAIN → READ → ARCHIVE. If the archive function could be called in a different context, for example,MAIN → SEND → ARCHIVE, that context would be represented by a different PCC value. However, in this simplified example, each function can only be called in one context.

For every caller PCC, there is an edge to each callee PCC. The transition probabilities can be found by taking the frequency of a given callee PCC divided by the sum of the frequencies for the corresponding caller PCC. For example, for the caller PCC MAIN, there are two possible callee PCCs. The probability of transitioning to MAIN → READ is $\frac{80}{80+20} = 0.80$. Alternatively, the probability of transitioning to MAIN → SEND is $\frac{20}{80+20} = 0.20$.

To calculate test suite adequacy, the sum of edges in the model observed during testing is divided by the sum of all edges in the model. For example, consider the example Markov chain shown in Figure 2.

If this Markov chain were constructed from user behavior, then when the application was in the hands of users, READ is called from MAIN 80% of the time while SEND is called 20% of the time. If during testing we exercised MAIN → READ,

then we would have a behavioral adequacy of:

$$\frac{.8}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .16$$

If instead, we tested MAIN → READ and READ → DELETE, then our adequacy increases because we are covering more code.

$$\frac{.8 + .35}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .23$$

However, if we test MAIN → READ and READ → ARCHIVE instead, then our adequacy increases further because ARCHIVE is more likely to be used than DELETE.

$$\frac{.8 + .55}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .27$$

This makes sense intuitively since testing more behaviors increases the score, and testing more frequently used behaviors further increases the score. Using this criterion, 100% adequacy is achieved when every behavior observed by the user-base is tested, and 0% is achieved when no behavior observed in the user-base is tested.

IV. Empirical Evaluation

To evaluate our proposed test suite adequacy criterion, we implemented a system for calculating behavioral adequacy for Android applications. The goals of the evaluation are as follows.

1) Determine the time and space overhead of the online behavioral profiling.
2) Determine the overhead associated with calculating behavioral adequacy offline.
3) Evaluate the behavioral adequacy of existing Android applications.

A. Experiment Design

To instrument applications, we used AspectJ because it provides a way to quickly instrument Java and Android applications. An AspectJ aspect was written to calculate the PCC value of the application at function calls. Only application defined functions were considered, so Android system calls and Java library calls were ignored. At each call, the current PCC, the caller, and the next PCC, the callee, were stored as a 64 bit value identifying a transition between PCC values. The frequency of these transitions were recorded, and written to a file upon an activity being paused, stopped, or destroyed. This data was then sent to a desktop PC for processing. A Java program was implemented to construct a Markov chain from the PCC edges. The test suite under study was then executed and the PCC edges collected on a per test basis. For structural coverage, Android's included EMMA tool was used.

Offline tasks were completed using a desktop running Centos 6.5 with a quad-core 1.6GHz CPU and 16MB of memory. User data was collected on an Asus Nexus 7 tablet running Android 4.3 and a Samsung Galaxy SIII smartphone running Android 4.1.1.

B. Case Studies

To conduct our evaluation, we selected several applications from the F-Droid open source appstore. We attempted to select well known applications with large test suites, however this was difficult since few applications contained test suites. The applications selected were K-9 Mail, and Github. K-9 Mail is an email application that can connect to IMAP, POP3, and SMTP servers to manage a user's email accounts. Github is an application that allows a Github user to interact with Github on an Android device. It supports browsing repositories, commenting, and creating Gists and issues.

Application	Files	Classes	Methods	Lines
K-9 Mail	230	806	5671	35410
Github	?	?	?	?

C. Metrics

We evaluate runtime overhead in terms of percent change in time. Space overhead in terms of percent change in source code occupied space on disk. Structural coverage is given in percent of code covered. Behavioral coverage is given as the sum of exercised edges over the sum of all edges.

D. Experimental Results

To evaluate online runtime overhead, the benchmarks' test suites were executed five times, and the execution time was measured with and without profiling instrumentation. The average of the five trials was taken. The table below shows the results, time is given in seconds.

Application	Time Uninstrumented	Time Instrumented	Percent Change
K-9 Mail	4.482	10.385	132
Github	66.006	70.445	7

The large difference in percent change between the applications warrants additional investigation. A possible explanation is that K-9's test suite primarily tests backend code that tends to complete very quickly, whereas Github's test suite involves testing UI elements, such as creating activities that is less sensitive to the instrumentation.

To evaluate space overhead, the benchmarks binary size was measured before and after instrumentation. Size is given in megabytes.

Application	Size Uninstrumented	Size Instrumented	Percent Change
K-9 Mail	2.91	3.35	15
Github	1.76	1.99	13

The size overhead between the two applications was about the same, with both applications using around 14% more disk space when instrumented.

To evaluate offline overhead, we measured the time needed to build the model from user data, and determine behavioral adequacy from the model.

To evaluate Android application test suites, we instrumented the benchmarks and allowed two users to interact with the applications for one day. Afterwards, we profiled the benchmarks' test suites, and constructed a model from the user data. We then calculated behavioral adequacy from the model and test data. For comparison, we determined the structural

coverage of the benchmarks' test suites using EMMA, and the PCC coverage by dividing the number of PCC transitions exercised by both the users and the tests and the number of PCC transitions exercised by the users.

Application	Behavioral Coverage	PCC Coverage	Method Coverage
K-9 Mail	0.00016	0.00024	7
Github	0.03824	0.04324	?

E. Threats to Validity

The most significant thread to the validity of our evaluation is the limited number of applications tested. The applications selected for the evaluation may not represent all Android applications. This problem can be alleviated by conducting a larger study on a wider range of applications. Another threat is the limited number of users participating in the study. The users participating in the study may not be representative of the rest of the user-base. The more the users interact with an application, the more likely they are to exercise PCC values not seen during testing, and thus decrease the score. This means users interacting with an application longer than normal will likely cause the behavioral adequacy to decrease. Additionally, users interacting with the application for less time than normal could cause the behavioral adequacy results to be too optimistic. Alternatively, since behaviors exercised by the test suite but not by the users are given a score of zero, users exercising very little of the application may cause the score to decrease. This issue can also be mitigated by a larger experiment with many users to increase the chance that the users represent an accurate sample of the larger user-base.

V. Related Works

Relative coverage is an alternative to traditional coverage that takes context into account when determining coverage. This is useful in software as a service systems where only a portion of a larger service is used by an application. From the perspective of the smaller application, some features the larger service provides are not used, and thus, irrelevant. These features do not need to be tested, and therefore should not be considered when determining coverage. Relative coverage excludes these unused feature from the coverage equation.

Miranda and Bertolino's work [7] on Social Coverage is the most similar to our work. They propose a system inspired by relative coverage that determines coverage according to context. Relative coverage systems rely on the developer to select which features are relevant, while social coverage, like us, uses observed user behavior to determine what features are important. Social coverage collects user data and can find similar users. The features used by these users might be relevant to the application. These features are taken into account when calculating social coverage.

The Synoptic system [8] also has similarities to our work. Synoptic is a tool that can infer finite state models from reading execution logs. Like us, they construct a model based on user behavior. However, synoptic requires the application log states, and is thus more suited to a high level model, whereas we model based on calling context. Additionally, we apply the learned model to test suite adequacy while Synoptic focuses on analysing logs.

The Gamma system presented by Orso et al. [9] attempts to enable remote monitoring of software after its deployment. Gamma does attempt to address the issue of runtime overhead, and allows for the costs of instrumentation to be shared among users. The developer can specify what type of information they are interested in, and the Gamma system divides the task of collecting this information among the userbase. Additionally, Gamma allows for its instrumentation to be modified by an update.

Bond and McKinley [5] introduced a technique for decreasing the costs of tracking a programs calling context called probabilistic calling context. This system allows a calling context to be represented as an integer that is easy to calculate and well suited to anomaly detection applications. The technique consists of a function that takes as input the current probabilistic calling context and an integer representation of the current context. It then outputs an integer representing the current probabilistic calling context. The function produces outputs that are uniformly distributed, so that the chance of conflict is low, and the order of the contexts is taken into account.

In a later work, Bond et al. [10] present a technique for calculating the entire calling context from the probabilistic calling context. This technique has the advantage of being able to reconstruct the calling context offline, however, some dynamic information is needed make a search of the context space feasible, which requires additional overhead of 10-20%.

Elbaum et al. [11] present a study showing how software evolution affects code coverage. The study shows how even small changes can have a large impact. This work is similar to ours in that we are concerned with the performance of structural adequacy criteria in evolving software environments.

Whittaker presented a series of papers [2], [3], [4] on using markov chains in software testing, including input generation, software specification, and statistical software testing.

Andrews et al. [12] present a paper analyzing the usefulness of mutation testing. The study shows that mutation testing creates faults similar to real faults. The abc compiler provides a way to perform AspectJ instrumentation on Android bytecode without access to the source code [13].

VI. Conclusion

Android applications exist in a rapidly changing environment. Traditional test suite adequacy criteria such as structural coverage and fault finding adequacy provide insufficient guidance to developers in such a rapid development cycle. As an alternative, we propose a behavior driven test suite adequacy criterion that can adapt to changes in the environment when assessing an applications test suite. By instrumenting behavior on applications running in the wild, a markov chain is constructed that models user behavior. This behavioral data is then compared with data obtained during the execution of a test suite to determine the test suites adequacy. A case study of two applications suggests that there is potential for major improvement in the quality of test cases for mobile applications. A more comprehensive empirical study is needed to explore the technique's run-time overhead and evaluate the adequacy of additional application's test suites.

ACKNOWLEDGMENT

This work is supported by NSF REU Grant 1359275.

REFERENCES

[1] Android, "Dashboards," http://developer.android.com/about/dashboards/index.html, Jul. 2014.

[2] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 1, pp. 93–106, Jan. 1993.

[3] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 10, pp. 812–824, Oct. 1994.

[4] J. Whittaker, "Stochastic software testing," *Annals of Software Engineering*, vol. 4, no. 1, pp. 115–131, 1997.

[5] M. D. Bond and K. S. McKinley, "Probabilistic calling context," *SIGPLAN Not.*, vol. 42, no. 10, pp. 97–112, Oct. 2007.

[6] J. G. Kemeny and J. L. Snell, *Finite markov chains*. van Nostrand Princeton, NJ, 1960, vol. 356.

[7] B. Miranda and A. Bertolino, "Social coverage for customized test adequacy and selection criteria," in *Proceedings of the 9th International Workshop on Automation of Software Test*, ser. AST 2014. New York, NY, USA: ACM, 2014, pp. 22–28.

[8] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, "Synoptic: Studying logged behavior with inferred models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 448–451.

[9] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: Continuous evolution of software after deployment," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 65–69.

[10] M. D. Bond, G. Z. Baker, and S. Z. Guyer, "Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses," *SIGPLAN Not.*, vol. 45, no. 6, pp. 13–24, Jun. 2010.

[11] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 170–.

[12] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 402–411.

[13] S. Arzt, S. Rasthofer, and E. Bodden, "Instrumenting android and java applications as easy as abc," in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Springer Berlin Heidelberg, 2013, vol. 8174, pp. 364–381.