Testing Android Applications Using Multi-Objective Evolutionary Algorithms with a Stopping Criteria

Anshuman Rohella*, Shingo Takada*

*Dept. of Information and Computer Science, Keio University, Yokohama, Japan

{anshuman.rohella, michigan}@doi.ics.keio.ac.jp

Abstract— **The ever increasing usage of Android devices and apps has created a demand for faster and reliable testing techniques. While the quality of test cases can be summed up based on the amount of code they cover, fault detection in applications is one of the main objectives for testing. We introduce an Android app testing approach which uses multi-objective genetic algorithm with elitism which finds optimal test cases by minimizing their length, maximizes the code coverage and fault detection capability, and minimizes the whole test suite for re-usability. In addition to that, we also incorporate a progress indicator which checks for improvements in test suite quality after subsequent generations and use it as a stopping criterion. The effectiveness of our approach is shown in our evaluation where it is able to perform better than the existing state-of-the-art tools.**

Keywords: Android Testing;Evolutionary Testing;Multi-Objective Testing

I. INTRODUCTION

The boom in the use of smartphones in personal and business related services has created a rapid increase in the demand for fast delivery of quality software. This demand has left Google Play Store with over 3.5 million apps, as of December 2017 [5] and this number is increasing every day. These apps are used by a huge number of people and to stay in business, the providers have to make sure that the apps are bug free and behave as planned.

Unlike a desktop application which runs as a single monolithic process, Android apps have to be a lot more flexible and a tightly coupled architecture among the components with event based control flow may make automated testing difficult and this causes heavy reliance on labour intensive manual testing [24]. Several attempts have been made in the past to automate Android Testing. Some of the techniques include random testing, model-based testing, testing with dynamic exploration etc.

Search-based software engineering techniques [15], [16] have been used in the past for GUI [11], [12], [14], as well as Android testing [10], [26] and have shown promising results. Even though code coverage is used as the basis of most of the testing tools, multiple objectives like fault detection, code coverage, shorter optimized test sequences have to be considered in order to generate optimal test suites.

DOI reference number: 10.18293/SEKE2018-084.

We propose an Android testing approach which uses an elitist based multi-objective genetic algorithm which covers all these objectives while exploring the application under test (AUT) and creates optimal test suites. In addition to that, since search-based testing requires execution of a large number of tests, a stopping criteria is required to stop the process with enough confidence to guarantee sub-optimal solutions. Most of the existing tools set a pre-defined upper bound on either the number of test cases executed or the number of iterations (generations) for which the process will run. This may result in either pre-mature convergence or waste of CPU time. We use a progress indicator, MDR [25] to calculate the improvement of every generation and stop the process if no improvement, or degradation is found.

The remainder of this paper is organized as follows: Section II provides related works on Android testing, Section III describes the methodology as well as the architecture, Section IV evaluates our approach, Section V outlines the limitations of our approach, and Section VI concludes the paper with the future work.

II. RELATED WORK

Google's Android Monkey [1], a random input generator for Android apps stress tests the app in order to achieve coverage and detect faults. Due to its integration with Android development toolkit, it is widely available, is easy to use and is therefore considered the state-of-the-art testing software for Android apps [8].

Dynodroid [30] is another state-of-the-art automated testing tool [8] which uses heuristics to explore the application. It is a bit "smarter" than Monkey since it uses frequency and relevance of events to generate test sequences. In addition, it allows the user to give input, for example login info, when the search is stalled.

EvoDroid [10] is one of the first search-based testing tools for Android and is closely related to our work. It uses a call graph (obtained using MoDisco [5]) and an interface model to generate a *Population* of test cases which achieve a fitness score based on unique paths/line in the code they cover in the call graph.

Sapienz [26] is another tool that combines fuzz testing and search based exploration for testing. It uses NSGA-II [34] to maximize coverage and fault revelation while decreasing the sequence length. It considers multiple levels of instrumentation and can be used for both white and black box testing.

III. Methodology

We use app's source code to generate two models. A *Screen Flow Graph* model and a *Widget-Screen Map* model. Details of these models are discussed later in this section. These two models are then used to create a population of Individuals (test cases). Finally, these test cases are then executed using the multi-objective algorithm and the process stops when the stopping criteria is met.

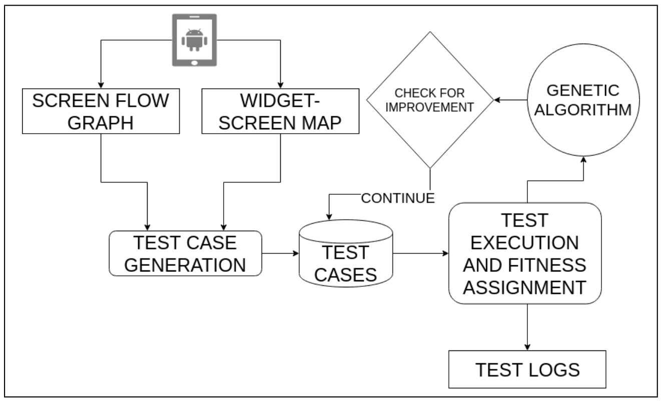

Fig. 1. The overall architecture

The overall architecture of our approach in shown in Fig. 1. Our approach can be divided into the following steps:

- Generate *Screen Flow Graph* to get the transition information of various activities, dialogues, options menu, etc.
- Generate *Widget-Screen Map* by obtaining the information about all the widgets/components available for user interaction on different screens.
- Generate test individuals based on the two models obtained earlier.
- Execute the test cases generated using the multi-objective algorithm.

A. Model Generation

In order to generate test cases, we generate the following two models:

- **Screen Flow Graph**: We used GATOR [29], an open source analysis toolkit which performs a static analysis on the app to capture the possible control/data-flow by tracking the event handling callbacks and window (Activities, dialogue boxes, listviews, etc) life cycle callbacks. This information is used to create a Screen Flow Graph, as shown in Fig. 2 , which is then used to create test cases with *sections* (discussed later) of events valid for a particular screen.
- **Widget-Screen Map**: In an Android app, each Activity is coupled with a layout XML file which contains organized information about that activity's components. These components include the type of layout (e.g., *LinearLayout, RelativeLayout, etc.*), and GUI interaction widgets known as *views* (e.g., *TextView, Button, ImageView, etc.*).
  Each view has a unique ID which is used in the application code's context. We first parse the Android-Manifest XML file to list all the activities in the app. To create the *Widget-Screen Map* model, each layout file attached to these activities/screens is parsed and the obtained widgets information is mapped to their respective screens. This information is used to create events for a particular screen.

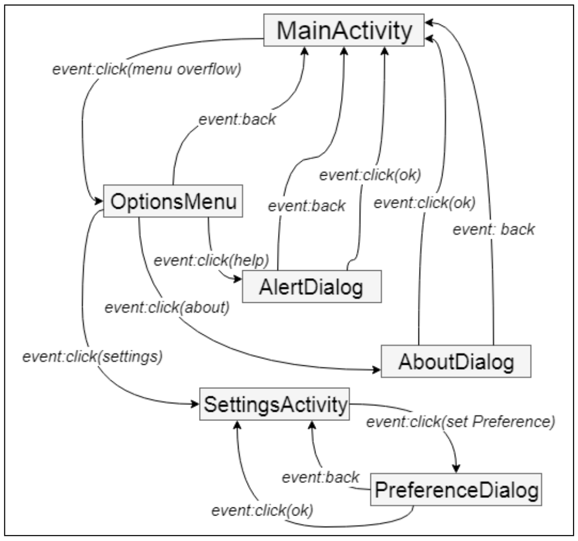

Fig. 2. An example of a Screen Flow Graph for an Android app. The nodes represent the various screens in the app and the edges represent the events that cause a transition.

We now discuss how these two models are used to create test individuals for the initial population.

B. Generating Test Individuals

Individual representation is an important aspect in evolutionary algorithms as it depicts how a single solution on the whole population is encoded. Prior researches like [11] and [23] have used entire test suites as individuals wherein each individual consists of a random number of test cases (genes). This approach can be ideal for test suite minimization. However, the overall quality of each gene (test case) remains the same throughout the process because the crossover and mutation takes place at test suite level and not the test case level. Therefore, we use individual representation similar to EvoDroid [10] where each test case represents a single individual in the population.

Fig. 3 shows an example of the representation of a single individual. To create an individual, the events for the launch screen (first screen shown to the user when the app starts) section are randomly selected from the *Widget-Screen Map* and are checked against the *Screen Flow Graph* for any transitions. If the selected event causes a transition to another screen, a new section is created for that screen and events are selected randomly for that screen. This process repeats itself until the number of sections reach *n_sec*, where *n_sec*

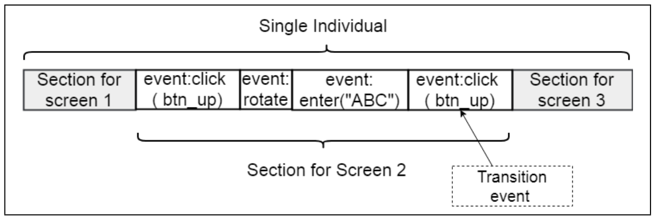

Fig. 3. Representation of a single Individual used in our approach.

is chosen randomly for each individual before its creation. If there is only one screen in the app, the test individuals contain only one section with *n_ev* random events, where *n_ev* is chosen randomly for each individual before its creation.

C. Evolutionary Approach

A number of multi-objective evolutionary algorithms [18], [19], [20], [21], [22], [34] have shown to be quite effective in solving problems with multiple constraints/objectives. We use CBGA-ES [22] genetic algorithm in our approach to optimize test cases by selecting shorter individuals that have high code coverage and high fault revelation capability. In addition, the algorithm selects *l* elite solutions after the first generation which further reduces the test suite as well as the search space for subsequent generations. The main idea of the approach is to cluster the individuals with similar fitness into *k* clusters, sort the cluster based on *cluster dominance strategy*, and then select the individuals(elite individuals) for the best cluster as a population for next generation. The details are as follows:

1) Clustering: After the execution of all the individuals in the first population, each individual gets a fitness vector containing its fitness value for each objective. For example, an individual I_a will have a fitness vector $F\_I_a = \{fval_1, fval_2, ..., fval_n\}$, where n is the number of objectives.

Next, k individuals are chosen randomly and their fitness vectors are assigned as the centres of the corresponding clusters. The CBGA-ES algorithm uses Lloyd's algorithm [31] to cluster all the individuals. The next step is to sort these k clusters using *cluster dominance strategy* [22].

2) Cluster Dominance Strategy: Consider two clusters C_a and C_b with centroids m_a and m_b, respectively. These two clusters are checked for either dominance or partial dominance:

- Cluster Dominance: $C_a \succ C_b$ (i.e., C_a completely dominates C_b) iff

$$\forall_{i=1-n} m_{ai} \leq m_{bi} \wedge \exists m_{ai} > m_{bi}$$

where m_{ai} and m_{bi} are the fitness values for the i^{th} objective in the two centroids, respectively.

- Cluster Partial Dominance: $C_a \succeq C_b$(i.e., C_a partially dominates C_b) if one of the two cases holds *true*.

Case 1.

$$n(\forall_{i=1-n} m_{ai} > m_{bi}) > n(\forall_{i=1-n} m_{ai} < m_{bi})$$

where $n(\forall_{i=1-n} m_{ai} > m_{bi})$ is the number of fitness values of centroid m_a which are better than those in m_b.

Case 2.

If $n(\forall_{i=1-n} m_{ai} > m_{bi}) = n(\forall_{i=1-n} m_{ai} > m_{bi})$,

$$\left(\Sigma_{i=1}^{n} \frac{m_{ai} - m_{bi}}{m_{ai}} \times 100\% \right) > 0$$

which means that for all the objectives, m_a is able to get higher percentage of centre values than m_b.

Once the clusters have been sorted, an *elite population* of size *l* is created by adding the individuals from the best cluster. If the size of the best cluster is less than *l*, the individuals from the next best cluster are added to the elite population. This elite selection strategy reduces the search space as compared to the initial population and helps create a minimized test suite with optimal test cases. This population then undergoes *crossover* and *mutation* and proceeds to the next generation unless the stopping criteria is met.

Crossover

Crossover is a genetic operator used in Evolutionary Algorithms to create individuals for the next generation. We use a single-point random crossover where two random points are chosen in the selected individuals and the crossover operation is performed. Test case level crossover increases the possibility of creating better test cases since the created individual contains events from the parent individuals. Fig. 4 shows an example of a crossover.

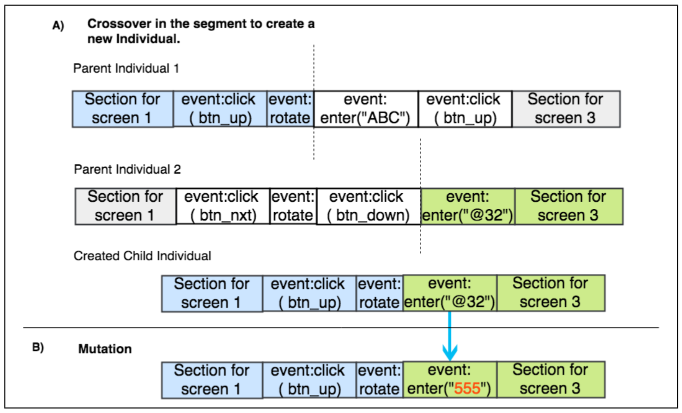

Fig. 4. Examples of crossover (A) and mutation (B).

To preserve the validity of the produced test case, crossover points are chosen in the sections which corresponds to the same screen (refer section B). A similar approach has been used in EvoDroid where they use multipoint probabilistic crossover in *segments* [10] belonging to the selected parent individuals.

Mutation

Mutation is another genetic operator used in Genetic Algorithms which mutates/changes a gene of an offspring with some probability to maintain genetic diversity in the population and avoid getting stuck in a local minima. In our case, we apply mutation at a gene (event) level by randomly selecting a segment in the new individual created and applying one of the following mutation operations.

- Adding/Removing: A new event valid for that segment is added/removed at a randomly selected position in the segment. Some screens may need a specific event to be performed for a specific number of times. This mutation may help an individual (test case) to cover those cases and possibly explore a new state.

- Changing the order: This mutation operator changes the sequence of events in a segment randomly. Events performed in a specific order may trigger a transition to a new state. For example, a game application may need button clicks in a specific order to advance to the next stage.

- Modifying gene properties: This specific mutation operation is only valid for events which need some sort of input (Text, Number inputs). For example, invalid/fuzzy inputs like blank spaces, number zero and special characters may not be handled properly by an app and may throw an exception.

Stopping Criterion

A genetic algorithm requires some criteria for termination. Some of the traditionally employed termination criteria include having an upper bound on the number of generations, giving the process a pre-defined time limit, and stopping the algorithm if there is no improvement in subsequent generations.

In our case, the number of generations and the initial population depends on the complexity of the problem. The chosen parameters may lead to either pre-mature convergence or CPU time wasted through executing redundant test cases with no improvement [9]. In addition to that, Android Emulators [2] are known to be slow and may further increase the total execution overhead.

According to previous researches [25], [27], [28], finding a stopping criteria for a multi-objective problem is a multi-objective problem itself since the improvement has to be considered for all the defined objectives. We use Mutual Domination Rate indicator (MDR) [25], a progress indicator that is specially designed for multi-objective evolutionary algorithms to check the quality of solutions after each generation. The details are as follows.

Consider PF^*_{t-1} and PF^*_t as the non-dominated (elite) solutions obtained at generation *t-1* and *t*, respectively. $I_{mdr}(t) \in [-1, 1]$ contrasts between the number of non-dominated individuals of generation *t* that dominate the non-dominated individuals of generation *t-1* and vice-versa.

Consider a function $C = d(A, B)$ that returns a set of elements of A that are dominated by at least one element of B, and $|C|$ is the number of elements in the set, then,

$$I_{mdr}(PF^*_t, PF^*_{t-1}) = \frac{|d(PF^*_{t-1}, PF^*_t)|}{|PF^*_{t-1}|} - \frac{|d(PF^*_t, PF^*_{t-1})|}{|PF^*_t|} \quad (1)$$

$I_{mdr} = 1$ indicates that generation *t* is completely better than *t-1*, $I_{mdr} = 0$ indicates that there has been no improvement since the last generation, and in the worst case, $I_{mdr} = -1$ indicates that the quality of the generation has degraded. If there is no improvement for a pre-defined number of generations, the execution process stops with the last test suite considered as the optimal test suite.

IV. EVALUATION

We establish and answer three research questions to compare our evaluation results with the study conducted by Choudhary et al. [8].

- **RQ1 (Code Coverage)**: How does the code coverage achieved by our approach compare to the state-of-the-art existing approaches as mentioned in study by Choudhary et al. [8]?
- **RQ2 (Fault Detection)**: How effective is our approach in finding faults?
- **RQ3 (Multiple Objective Handling)**: How effective is our approach in handling the trade-off between the three objectives used for evaluation?

Creating a test oracle is another problem in the field of automated testing and has been covered in various researches [32], [33]. However, our main focus here is not the behaviour of the application or validity of the test cases, but to increase the code coverage, detect faults, reduce the test sequence length and minimize the test suite.

A. Experimental Environment and Settings

All our experiments were done on Nexus 5's emulator with 1586 MB of RAM. The emulator was run on a 64-bit MacOS 10.12.4 machine with 2.5 GHz Intel Core i5 processors and 8 GB of RAM. We ran the evalution on a set of open source Android applications. We set the initial population *n* to 100 test cases and the elite population l to 50 and the number of clusters *k* to 2. The mutation probability was 0.2 with 50% of the elite solutions undergoing crossover (25 solutions in our case). Since our test cases represents an individual with varying number of events in it, they cannot be compared directly to the event inputs used in DynoDroid and Monkey in terms of number of test cases executed. Hence, we evaluated our approach first, and the time spent on each app using our approach was then given to DynoDroid and Monkey to compare the performance. Since our approach is non-deterministic, we ran the evaluation 10 times and took an average of the results. We used JACOCO [4] to obtain the code coverage and all the test cases were in Espresso [3] and UI Automator [6] format.

TABLE I

ACCUMULATED CODE COVERAGE FOR OUR APPROACH (#COVE), MONKEY(#COVM) AND DYNODROID(#COVD) WHERE LOC IS THE LINES OF CODE IN THE APP AND THE EXECUTION TIME, TIME IS IN MINUTES.

AUT	LOC	#CovE	#CovM	#CovD	Time
Munchlife v1.4.2	163	93.86%	58%	76%	112
Munchlife v1.4.4	186	94.62%	63.55%	74.82%	101
BatteryCircle	251	82%	62%	79.38%	34
Triangle	281	91%	69%	81.23%	38
JustSit	276	75%	43%	66%	102
CalorieMate v1.0.0	132	84%	53%	81.22%	109
CalorieMate v1.1.0	197	87.30%	67.65%	80.27%	119
TippyTipper	996	88%	81.54%	51.33%	125
LearnMusicNotes	398	62%	50.7%	47%	117
Bats-HIIT	316	40%	24%	45%	89
BatteryDog	466	64%	51%	62.54%	34
SpriteText	1165	60.51%	58.21%	58.76%	48
SwiFTP	2160	22.8%	12.44%	16.22%	92
PasswordMaker	1436	42.96%	22%	32.22%	56
Translate	711	37.97%	19.44%	32%	78

TABLE II

FAULTS DETECTED DURING THE EVALUATION.

AUT	Faults	Exception
JustSit	1	java.lang.SecurityException
Bats-HIIT	1	java.lang.NullPointerException
CalorieMate v1.0.0	2	java.lang.IllegalArgumentException
		java.lang.NumberFormatException

B. Evaluation Results and Discussion

We chose a set of 15 (2 of which were different versions of the respective apps) open source apps to compare the line coverage of our approach, Monkey, and DynoDroid.

RQ1(Code Coverage): As shown in Table I, our approach is able to achieve a a higher coverage than Monkey and DynoDroid for most of the apps.

Some of the reasons for not achieving higher code coverage are due to dependence on external native apps (e.g., camera, contacts, messaging) and unavailability of external services on an Emulator (e.g., the app *SwiFTP* requires a WiFi connection to start a FTP server). Also, some apps may have asynchronous behaviour based on timers/thread events. For example, apps like *Bats-HIIT* (exercise timer) and *LearnMusicNotes* have features where a new state is created (register high score/personal best) only after the timer runs out. These events cannot be handled unless a specific timeout is defined, regardless of the technique used.

RQ2(Fault Detection): The fault finding capability of our approach is shown in Table II

Our approach was able to find a total of four new exceptions which lead to app crashes. These issues were reported to the app's respective online repositories. These exceptions were not detected by Monkey or DynoDroid during our evaluation.

RQ3(Multiple Objective Handling): To answer the third research question, we compared the first generation and the last generation in terms of decrease in average length of the fault revealing test cases for all the apps in Table II.

TABLE III

DECREASE IN THE LENGTH OF FAULT REVEALING TEST CASES.

AUT	Decrease %
JustSit	74%
Bats-HIIT	82.21%
CalorieMate v1.0.0	64.12%

TABLE IV

EXECUTION TIME AND COVERAGE RESULTS FOR OPTIMAL TEST SUITES.

App Name	Coverage%	Time
JustSit	75%	16
Bats-HIIT	42%	15
CalorieMate v1.1.0	74.12%	24

The results are shown in Table III. This shows the effectiveness of our approach in choosing test sequences of shorter length if the same faults are found by multiple test cases. In addition, to check the re-usability of the optimized test suite, we ran the optimal test suites on newer versions/fixed versions of the apps. The coverages obtained on a single run are shown in Table IV.

The effectiveness of the optimal test suites on newer versions of the apps depends on the type of changes made. The test suites may have limited coverage if new features like widgets/functions are added. For example, new views were added in *CalorieMate v1.1.0* and therefore the coverage obtained is less than the coverage obtained when the app was used for normal execution (Table I).

C. Threats to Validity

Two important threats to validity are the following:

- First, even though the evaluation apps were chose at random, our results cannot be generalized since our technique may not be applicable to all type of apps.
- Second would be the type of stopping criteria used. Even though the MDR improvement index has been found to be effective in our implementation, it still checks the improvement of the search locally, i.e., comparing t^{th} generation with $t-1^{th}$ generation. This may limit the whole search to a local optima.

V. LIMITATIONS OF OUR APPROACH

This section outlines the current limitations of our tool.

A. Static Analysis Limitations

Currently, our approach is limited to the scope of the static analysis techniques available for Android. As discussed in GATOR [29], modeling transitions caused by asynchronous events (timers and sensors) is difficult and is yet to be handled. This can lead to an incomplete model.

B. Limited Information in Layout

Depending on the way the app is written, complete information about a View may not be obtainable. For example, a developer may choose to have views without specifying their resource ID, or, options in a ListView may be added in through code (sometimes dynamically while the user exercises the app) rather than defining them in resource XML files. A dynamic modeling/assertion may handle these cases in a better way.

VI. Conclusion and Future Work

We proposed an approach to test Android applications which uses CBGA-ES algorithm to maximize code coverage, find short fault revealing test cases and minimize test suite for possible re-usability. In addition, we incorporate a stopping criterion to limit the testing time in case no improvement is detected over the generations. Our approach was able to achieve significantly higher coverage than current state-of-the-art tools. Also, we were able to use the minimized test suite for newer versions of three apps.

Future work includes incorporating better model generation to our approach to increase the variety of applications to which our approach can be applied to. Also, it is possible to check improvement on a global level rather than checking for local improvements after every generation. This can be done by analyzing the complexity of the application and/or predicting a global pareto front by analyzing the local improvements using MDR [25].

References

[1] Android Monkey: https://developer.android.com/studio/test/monkey.html

[2] Android Virtual Device: https://developer.android.com/studio/run/managing-avds.html

[3] Android Espresso : https://developer.android.com/training/testing/espresso/index.html

[4] JACOCO: http://www.eclemma.org/jacoco/

[5] Total Apps on Google Play: https://www.statista.com/statistics/266210/number-of-availableapplications-in-the-google-play-store/.

[6] UI Automator: https://developer.android.com/training/testing/ui-automator.html

[7] MoDisco: https://www.eclipse.org/MoDisco/

[8] S. R. Choudhary, A. Gorla and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?," 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, 2015, pp. 429-440.

[9] Z. Li, M. Harman and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," IEEE Transactions on Software Engineering, vol. 33, no. 4, pp. 225-237, April 2007.

[10] R. Mahmood, N. Mirzaei, and S. Malek. 2014. "EvoDroid: segmented evolutionary testing of Android apps," Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 599-609.

[11] F. Gross, G. Fraser, and A. Zeller. 2012. "EXSYST: search-based GUI testing," Proc. of the 34th International Conference on Software Engineering (ICSE '12). IEEE Press, Piscataway, NJ, USA, 1423-1426.

[12] F. Gross, G. Fraser, and A. Zeller. 2012. "Search-based system testing: high coverage, no false alarms," Proc. of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012). ACM, New York, NY, USA, 67-77.

[13] K. Inkumsah and T. Xie, "Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution," 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, 2008, pp. 297-306.

[14] S. Carino and J. H. Andrews, "Dynamically Testing GUIs Using Ant Colony Optimization," 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, 2015, pp. 138-148.

[15] M. Harman. 2007. "The Current State and Future of Search Based Software Engineering," 2007 Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA, 342-357.

[16] M. Harman, S. Afshin Mansouri, and Y. Zhang. 2012. "Search-based software engineering: Trends, techniques and applications," ACM Comput. Surv. 45, 1, Article 11 (December 2012), 61 pages.

[17] M. Harman, S. Afshin Mansouri, and Y. Zhang. "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Department of Computer Science, King's College London, Tech. Rep. TR-09-03 (2009).

[18] E. Ziztler, M. Laumanns, and L. Thiele. "SPEA2: Improving the strength Pareto evolutionary algorithm for multiobjective optimization," Evolutionary Methods for Design, Optimization, and Control (2002): 95-100.

[19] J. Knowles and D. Corne, "The Pareto archived evolution strategy: a new baseline algorithm for Pareto multiobjective optimisation," Proc. of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406), Washington, DC, 1999, pp. 105 Vol. 1.

[20] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. , "MOCell: A cellular genetic algorithm for multiobjective optimization," Int. J. Intell. Syst.(2009), 24: 726–746.

[21] E. Zitzler, K. Deb, and L. Thiele. "Comparison of multiobjective evolutionary algorithms: Empirical results," Evolutionary computation 8.2 (2000): 173-195.

[22] D. Pradhan, S. Wang, S. Ali, T. Yue and M. Liaaen, "CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, 2017, pp. 367-378.

[23] G. Fraser and A. Arcuri. 2011. "EvoSuite: automatic test suite generation for object-oriented software," Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 416-419.

[24] M. E. Joorabchi, A. Mesbah and P. Kruchten, "Real Challenges in Mobile App Development," 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, 2013, pp. 15-24.

[25] J. L. Guerrero, J. Garcia, L. Marti, J. Manuel Molina, and Antonio Berlanga. 2009. "A stopping criterion based on Kalman estimation techniques with several progress indicators," Proc. of the 11th Annual conference on Genetic and evolutionary computation (GECCO '09). ACM, New York, NY, USA, 587-594.

[26] K. Mao, M. Harman, and Y. Jia. 2016. "Sapienz: multi-objective automated testing for Android applications," Proc.s of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). ACM, New York, NY, USA, 94-105.

[27] H. Trautmann, T. Wagner, B. Naujoks, M. Preuss and J. Mehnen, "Statistical Methods for Convergence Detection of Multi-Objective Evolutionary Algorithms," Evolutionary Computation, vol. 17, no. 4, pp. 493-509, Dec. 2009.

[28] L. Martí, J. García, A. Berlanga and J. M. Molina, "A progress indicator for detecting success and failure in evolutionary multi-objective optimization," IEEE Congress on Evolutionary Computation, Barcelona, 2010, pp. 1-8.

[29] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan and A. Rountev, "Static Window Transition Graphs for Android," 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, 2015, pp. 658-668.

[30] A. Machiry, R. Tahiliani, and M. Naik. 2013. "Dynodroid: an input generation system for Android apps," Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). ACM, New York, NY, USA, 224-234.

[31] S. Lloyd, "Least squares quantization in PCM," IEEE Transactions on Information Theory, vol. 28, no. 2, pp. 129-137, March 1982.

[32] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 507-525, May 1 2015.

[33] G. Fraser and A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles," IEEE Transactions on Software Engineering, vol. 38, no. 2, pp. 278-292, March-April 2012.

[34] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, Apr 2002.