AGRippin: A Novel Search Based Testing Technique for Android Applications

Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana
Department of Electrical Engineering and Information Technologies
University Federico II of Naples
Via Claudio 21, Naples, Italy
{domenico.amalfitano, nicola.amatucci, anna.fasolino, porfirio.tramontana}@unina.it

ABSTRACT

Recent studies have shown a remarkable need for testing automation techniques in the context of mobile applications. The main contributions in literature in the field of testing automation regard techniques such as Capture/Replay, Model Based, Model Learning and Random techniques. Unfortunately, only the last two typologies of techniques are applicable if no previous knowledge about the application under testing is available. Random techniques are able to generate effective test suites (in terms of source code coverage) but they need a remarkable effort in terms of machine time and the tests they generate are quite inefficient due to their redundancy. Model Learning techniques generate more efficient test suites but often they do not not reach good levels of coverage. In order to generate test suites that are both effective and efficient, we propose in this paper AGRippin, a novel Search Based Testing technique founded on the combination of genetic and hill climbing techniques. We carried out a case study involving five open source Android applications that has demonstrated how the proposed technique is able to generate test suites that are more effective and efficient than the ones generated by a Model Learning technique.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Android, Search Based Testing, Genetic Algorithms

1. INTRODUCTION

The diffusion of smartphones and other mobile devices has grown exponentially in the last years with a corresponding growth of the number and of the complexity of the developed applications. These applications are often realized in a very rapid way, with high pressure due to a very small time-to-market. Moreover, most of the applications are developed by small teams or single developers, that devoted very limited time and resources to design and testing activities. A recent study of Kochhar et al. [14], based on open source Android projects hosted on GitHub, has shown that practices related to testing automation are rarely diffused. In particular, they found that only 14% of the apps they have analyzed contains executable test cases and only 4 apps out of 627 have test cases able to cover more than 40% of the source code of the applications. More generally, Muccini et al. [20] have stated the need for testing automation techniques applicable to the context of mobile applications. Joorabchi et al. [13] on the basis of interviews to 12 senior mobile app developers have concluded that the practice of manual testing is nowadays prevalent for mobile applications and that the automation of GUI testing remains a challenging task.

The main techniques proposed in literature to automate test cases generation and execution in the context of mobile applications can be classified in Capture/Replay, Model based, Model Learning and Random techniques [8].

Capture/Replay techniques record user interactions generated by human testers and convert them into test scripts that can be automatically replayed. Recent contributions in this field are the ones of Liu et al. [15] and White et al. [24]. Capture/Replay techniques need a remarkable effort to collect a sufficient number and variety of interactions from users or testers in order to obtain effective test suites. Moreover, test cases produced by different testers may be very similar between them, so the test suites may be quite redundant and inefficient. Model Based techniques are able to generate test cases from structural and/or behavioural models of the application. Examples of model based techniques in the field of mobile applications are the ones proposed in [23], [12], [26] and [4]. Model Learning techniques are able to build their own models by systematically exploring the behaviour of the GUI of a mobile application. They automatically generate test cases corresponding to sequences of triggered user and system events. In the context of mobile applications, recent Model Learning techniques have been proposed in [6], [17], [5]. Random testing techniques differ from Model Learning techniques because the sequence of triggered events is randomly chosen. Random techniques are very popular in the Android environment. In particular, Monkey [1] is a command line executable testing tool included

[1] http://developer.android.com/tools/help/monkey.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DeMobile'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3815-8/15/08...$15.00
http://dx.doi.org/10.1145/2804345.2804348

in the standard Android Development Toolkit (ADT) that is able to generate sequences of events on the interface of an Android application in a totally automatic manner. Other random techniques have been recently proposed in [10], [16] and [3]. The popularity of random techniques is due to their suitability to every application without the need for any specific knowledge about them and to the good level of effectiveness that they are able to reach. Unfortunately, random techniques generate test cases that are redundant, inefficient and very difficult to comprehend and manage (e.g. for debugging purposes).

Search based testing techniques represent a promising trade-off between Model Learning and Random techniques to generate effective and efficient test suites. Search Based Software Testing [1] is a specialization of Search Based Software Engineering (SBSE) [11] [9] related to the application of metaheuristic techniques to the problem of automatic generation of test cases optimizing the fault finding or the code coverage with a reasonable effort. In particular, in the set of metaheuristics algorithms, genetic algorithms are often used [2]. Genetic algorithms try to imitate the natural process of evolution: a population of candidate solutions, called chromosomes (i.e. test cases) is evolved using search operators such as selection, crossover, and mutation, gradually improving the fitness value of the individuals, until an optimal solution has been found or the search is stopped after a fixed time or a fixed number of evolutions.

At the best of our knowledge, only a single contribution related to the application of Search Based techniques to mobile application GUI testing can be found in literature. Mahmood et al. [17] present a search based technique supported by the EvoDroid tool for evolutionary testing of Android applications. EvoDroid automatically extracts two static models of the application under test, i.e. the Interface Model and the call Graph Model and generates evolutionary tests on the basis of these models.

In this paper we propose a novel search based testing technique called AGRippin (that is an acronym for Android Genetic Ripping) applicable to Android applications with the purpose to generate test suites that are both effective in terms of coverage of the source code and efficient in terms of number of generated test cases. Our technique is based on the combination of genetic and hill climbing algorithms and presents some peculiar contributions. In details, we propose:

- a representation based on a GUI test model that is able to describe test cases in form of *chromosomes* and actions on GUI interfaces in form of *genes*;
- a single cut crossover technique that is able to generate re-executable test cases;
- a mutation technique based on input equivalence classes;
- a fitness metric based on the measure of the diversity between the code coverage provided by the test cases;
- a technique to combine the genetic algorithm with a hill climbing algorithm in order to increase the rapidity of the algorithm.

The proposed technique has been implemented by a tool extending our previously presented Android Ripper tool [5] and tested in a case study involving five open source applications published on Google Play. Our approach differs from the EvoDroid one because it is completely automated and it does not rely on any previous knowledge or model of the application under test.

The paper is organized in the following way: section 2 reports a description of the the proposed technique while section 3 shows the results obtained of the case study. Finally, conclusive remarks and future works are proposed in section 4.

2. THE AGRIPPIN TECHNIQUE

In this section AGRippin, a Search Based technique for the generation of test cases optimizing the effectiveness in terms of coverage of the source code of the applications under test, is described.

According to the terminology of genetic algorithms, the *solution* proposed by the algorithm is, at each iteration, an evolved test suite that is composed of a *population* of *chromosomes* corresponding to test cases. Each chromosome is composed of *genes* corresponding to basic interactions with the application under test (AUT).

The effectiveness η of a test suite T is measured (in percentage) as the fraction of lines of source code (LOCs in the following) of the AUT covered by at least one of the test cases composing the test suite generated by the algorithm. It can be evaluated by the following formula:

$$\eta(T) = 100 * \frac{|\bigcup_{t \in T} Cov(t)|}{|LOC|}$$

where $t \in T$ is a test case included in the test suite T, $Cov(t)$ is the set of lines of code that is covered by the test case t and LOC is the set of lines of code of the AUT.

The efficiency ϵ of a test suite T can be defined as the ratio between its effectiveness and the number of generated test cases:

$$\epsilon(T) = \frac{\eta(T)}{|T|}$$

Our technique adopts a constraint of genetic algorithms for which the size of the population is constant at each iteration and is equal to the size of the initial population. Due to this constraint, the maximization of the effectiveness implies the maximization of the efficiency.

In the next subsection are described the characteristics of our technique in terms of chromosome representation, metrics for fitness evaluation, techniques for crossover, mutation, selection, and combination with a hill climbing technique.

2.1 Representation

The test suites generated by our technique are composed of test cases that are sequences of interactions with the GUI of the AUT. The application GUIs are abstracted according to the conceptual model shown in Figure 1.

According to this model, the GUI is composed of instances called *GUI Interfaces*; a GUI Interface is composed of a set of visual items called *Widgets*; each Widget is defined by a *Type* and some *Properties* with their *Names* and *Values*. Examples of Widget properties are its position on the screen, its identifier and so on. *Event Handlers* are methods that can be defined in the context of a GUI Interface or directly in the context of a Widget and that are executed in response to the occurrence of an *Event*. Events may be *User Events* if they are triggered by a user interaction on the GUI (e.g.

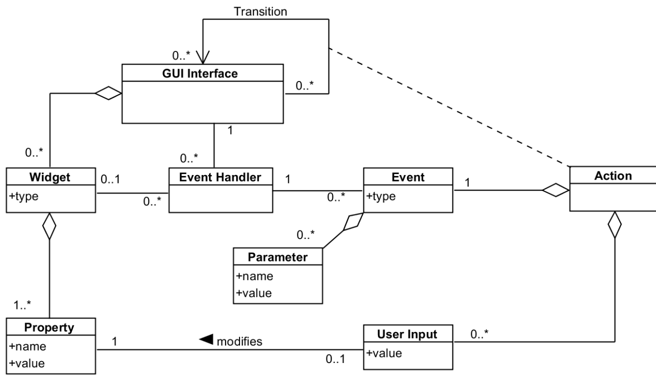

Figure 1: Conceptual Model of a GUI Interface

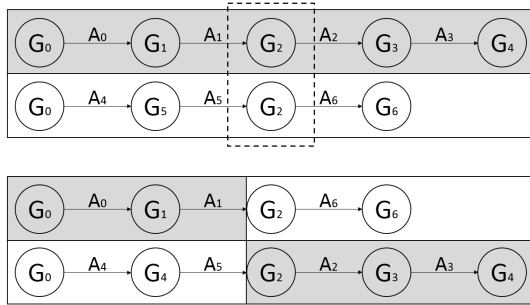

Figure 2: Crossover Example

the tap on a button), or *System Events* if they are triggered by the execution environment (e.g. a pausing of the application). An Event may have zero or more *Parameters*; each Parameter is identified by a *Name* and a *Value*. As an example, parameters of a tap event are the coordinates of the point of the GUI Interface where the tap is performed by the user. An *Action* is composed by an Event and one or more *User Inputs* and is able to trigger a *Transition* between two GUI Interface instances (not necessarily different between them). An User Input consists of the modification of a Value of a property (e.g. the insertion of a text in a editable text field) that does not cause the execution of any event handler (elsewhere it is modeled as an Action).

On the basis of these definitions, a *Test Case t* is a sequence of pairs $\{G, A\}$, where A is an Action that can be performed on the GUI Interface G and that may generate the GUI Interface of the next pair of the sequence. The first pair starts from the Home interface of the app G_0. A Test Case can be also defined as $t = (G_0, A_0, ... G_m, A_m)$, where G_0 is the Home interface. A set of such sequences is a *Test-Suite*.

2.2 Crossover

The crossover operator we exploited in our implementation is a Single-Point Crossover. Given two Test-Cases t_1 and t_2, the operator randomly chooses two pairs $\{G_i, A_i\} \in t_1$ and $\{G_j, A_j\} \in t_2$ and operates the crossover operation as shown in Figure 2.

A problem of this crossover operator is that it may generate sequences that do not correspond to executable test cases. If the generated test cases cannot be executed, they have to be discarded and the crossover operator has to be repeated until it generates a pair of executable test cases. In order to reduce the occurrence of such non-executable test cases, we propose a technique to candidate pairs of test cases and cut points for which the crossover operator should be applicable, based of two heuristic criteria of equivalence between GUI interfaces and between actions. The two heuristic criteria of equivalence are defined in the following ways:

EC1 Two GUI interfaces are considered equivalent if they include the same set of widgets and they define the same set of event handlers.

EC2 Two actions are considered equivalent if they are associated to the same user actions and the same event.

Let's consider two test cases $t_1 = (G_0, ..., G_i, A_i, ...)$ and $t_2 = (G_0, ..., G_j, A_j, ...)$ having the same starting GUI interface G_0. The pairs (G_i, A_i) and (G_j, A_j) are a candidate crossover point for our heuristic technique if they satisfy all these four criteria:

C1 the two GUI interfaces G_i and G_j are equivalent according to the $EC1$ criterion;

C2 the two actions A_i and A_j are not equivalent according to the $EC2$ criterion;

C3 the subsequence of t_1 which precedes the GUI interface G_i and the subsequence of t_2 which precedes the GUI interface G_j are not composed of a sequence of GUI interfaces and actions that are all respectively equivalent (according to the two criteria $EC1$ and $EC2$), *and* they are not both empty;

C4 the subsequence of t_1 which follows the action A_i and the subsequence of t_2 which follows the action A_j are not composed of a sequence of GUI interfaces and actions that are all respectively equivalent (according to the two criteria $EC1$ and $EC2$), *and* they are not both empty.

It's interesting to note that the first criterion avoids to select crossover points for which the action A_j is not applicable to the GUI interface G_i or the action A_i is not applicable to the GUI interface G_j. The other three criteria avoids the selection of crossover points that generate two test cases that are too similar or identical to the original ones.

As an example, let's observe the crossover example in Figure 2, in which equivalent GUI interfaces are labeled with the same label. We can verify that the selected crossover point (corresponding to the pairs (G_2, A_2) and (G_2, A_6)) is the unique one that satisfies all the four criteria (the pairs (G_0, A_0) and (G_0, A_4) satisfy the first two and the fourth criterion but they do not satisfy the third criterion because they are both preceded by an empty sequence).

The crossover points are randomly chosen in the set of the ones that satisfy these criteria. The test cases t_1 and t_2 are not removed from the test suite after the execution of the crossover operator, in coherence with the techniques proposed in the steady state genetic algorithms [22] (in example the Genitor one proposed by Whitley [25]). These techniques cause an increase in the population size that is restored to its initial size by the selection operator that is presented in the following. The crossover operator may be executed multiple times in the same iteration of the algorithm. We define the *crossover ratio* as the ratio between the number of test cases generated by the crossover at each iteration and the number of test cases of the initial solution.

2.3 Mutation

The mutation operator we proposed in this technique modifies the Actions by mutating the values of the User Inputs or the Event Parameters values. In each mutation, the value of a single parameter of the action is changed to a new value belonging to a static set of equivalence classes according to the parameter type. In example, the value of an editable text field may be set to a random string, a number or a correct email address while the location parameter of a GPS event may be set to coordinates values over or under the equator. The new test obtained after a mutation could not be executable if the GUI interface reached after the execution of the mutated action is not equivalent to the one reached by the original action. In this case, we consider that the new mutated test case terminates with the mutated action and the new test case is shorter than the original one.

The mutation operator randomly selects the test case and the action to be mutated in all the test suite. Mutated test cases are added to the test suite and the original ones are not removed. We define the *mutation ratio* as the ratio between the number of test cases generated by the mutation operator and the number of test cases of the initial solution.

2.4 Fitness Evaluation

We defined two distinct Fitness measures: the Global Fitness (that is the effectiveness η of the generated test suite and is measured in terms of the code coverage reached by the test cases of the test suite as described above in this section), and the Local Fitness expressing the degree of diversity of a single test case with respect to the set of the test cases of the test suite.

The Local Fitness measure ranks the individuals in terms of their potential contribution to the Global Fitness of the solution and of their diversity. To this aim, we propose a *rank* measure that is able to order all the test cases of the test suite. The Local Fitness measure is composed of two components named $F1$ and $F2$. The first component $F1$ the following three values, in order of decreasing rank:

- L_1, if the test case covers one or more lines that are not covered by any other test case;
- L_2, if the test case has a coverage set that (i) includes only lines of code that are covered by at least another test case of the test suite but that (ii) is not included in the set of lines covered by any other test case of the current test suite;
- L_3, if the test case has a coverage set that is included in the coverage set of at least another test case of the solution.

As regards the set of test cases having the same coverage set, the algorithm conventionally assigns a L_2 value to one test case (randomly selected) of the set and the L_3 value to all the other test cases of the set. Intuitively, test cases having a L_1 value are the ones that should be preserved to avoid a sure loss of effectiveness, whereas test cases having a L_3 value are the better candidates to be filtered out by the selection operator.

The second component $F2$ of the Local Fitness represents a weighted measure of code coverage and is defined by the following formula:

$$F2(t) = \sum_{l \in Cov(t)} w(l)$$

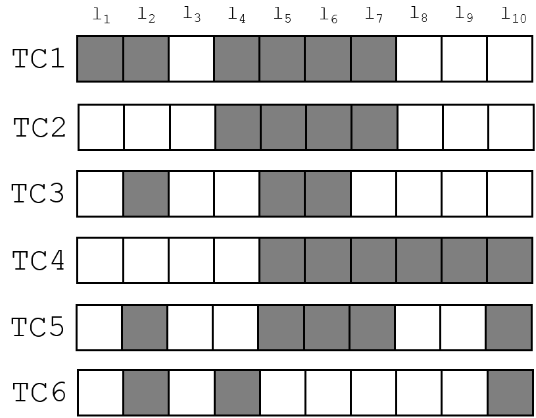

Figure 3: Test-Case Fitness Evaluation

where:

- $Cov(t)$ is the set of lines of code that are covered by the test case t
- $w(l)$ represents the relative weight of the coverage of the line l. It is defined as:

$$w(l) = \frac{1}{\sum_{u \in T} c(u)}$$

where $c(u) \in \{0, 1\}$. It is 0 if $l \notin Cov(u)$, 1 elsewhere.

$F2$ gives a measure of the relative importance of the coverage provided by a test cases in the context of a test suite because the coverage of lines that are covered by few test cases has a higher weight than the coverage of lines covered by many test cases. $F2$ is used to order between them test cases having the same $F1$ values.

As an example, let's consider the test suite shown in Figure 3 in which there is an AUT composed of 10 LOCs labeled as $\{l_1, ..., l_{10}\}$ and a Test Suite T = $\{TC1, ..., TC6\}$. The coverage of each test case is depicted in Figure 3 where black boxes corresponds to covered lines whereas white boxes corresponds to uncovered lines. In order to evaluate the Local Fitness we assigned $F1(TC1) = L_1$ and $F1(TC4) = L_1$ because they are respectively the unique test cases covering the line l_1 and the two lines l_8 and l_9. The values of $F1(TC2)$ and $F1(TC3)$ are instead set to L_3 because their coverage sets are respectively included in the ones of $TC1$ and $TC5$. The $F1$ value of the remaining test cases (i.e. $TC5$ and $TC6$) is set to L_2. In order to evaluate the $F2$ values for each test case, the weights w of each line l have to be evaluated. In example, the weight of line l_1 is 1 because it is covered exactly by one test case, whereas $w(l_2) = \frac{1}{4}$ because the line l_2 is covered by four test cases and so on. The Fitness Function of the test case $TC1$ is then equal to:

$$F2(TC1) = w(l_1) + w(l_2) + w(l_4) + w(l_5) + w(l_6) + w(l_7) = \frac{1}{1} + \frac{1}{4} + \frac{1}{3} + \frac{1}{5} + \frac{1}{5} + \frac{1}{4} = 2.23$$

The $F2$ values for each test case are reported in Table 1. In this table the test cases are ordered for decreasing values of $F1$ and, for test cases with the same $F1$ value, for decreasing values of $F2$. The $RANK$ column expresses the ordering position between all the test cases of the test suite.

Table 1: **Test-Case Classification Example**

RANK	**t**	$F1$	$F2$
1	TC4	L_1	2.98
2	TC1	L_1	2.23
3	TC5	L_2	1.23
4	TC6	L_2	0.91
5	TC2	L_3	0.98
6	TC3	L_3	0.65

2.5 Selection

The selection operator restores the size of the test suite to its initial value (corresponding to the size of the initial test suite) by deleting the test cases having the worst values of Local Fitness in coherence with the rank selection operator firstly proposed by Baker [7].

The fraction of test cases that are selected for deletion at each iteration is named *turnover ratio* and it is the sum of the *crossover ratio* and of the *mutation ratio*. As an example, if the crossover ratio is 1/3, then the test cases TC2 and TC3 shown in Table 1 have to be deleted.

2.6 Combination Technique

By means of the application of the crossover and of the mutation operator GUI interfaces that are not equivalent to any of the already visited ones may be discovered. These GUI interfaces have not yet been visited by any test case and they contains different sets of widgets and event handlers with respect to the other ones. This represents a positive achievement in terms of global fitness because code that has not yet been covered and that corresponds to the execution of these event handlers may now be executed.

In the AGRippin technique we propose a *combination* of the genetic technique with a model learning technique that will be started only when a new GUI interface is discovered. This technique aims at the systematic generation of new test cases including at least an event of each new discovered GUI interface and is very similar to the one we have proposed in the past [5]. This technique can be seen as a Hill Climbing technique because it selects at each iteration the most promising sequences, i.e. the ones in which at least a new line, corresponding to a new event handler call is covered. In order to restore the size of the test suite to its initial value, a re-execution of the selection operator has to be carried out after each execution of the model learning technique.

The adoption of hybrid algorithms combining genetic and hill climbing algorithms have been already presented in literature, with good results [18] and some criticism. We adopted this solution for two reasons: (1) because our specific implementation of the mutation operator is not able to generate new events but only to mutate their parameters and (2) to accelerate the process of exploring the interactions related to portions of the application that are discovered but not explored by crossover and mutation operators.

3. CASE STUDY

This section reports the results of some case studies that we carried out with the aim to assess the effectiveness of the proposed search based testing technique. We have implemented the technique in the context of Android applications and we have applied it to five open-source Android applications.

We have compared the test suites generated by our technique with the ones generated by the Android Ripper Tool [5] that we developed in the past. It realizes a Model Learning technique for the exploration of the GUI of Android applications. Since the Android Ripper explores at each iteration the GUIs of an Android application by executing an event that have never been executed before, we can consider this technique as a kind of Hill Climbing technique because an increment in code coverage is surely expected by the execution of each new event.

The purpose of our experimentation is to provide an answer for the following research question:

RQ: Are the test suites generated by the proposed technique more effective than the ones generated by the considered Hill Climbing technique?

The effectiveness of the generated test suites is measured (in percentage) as the fraction of lines of code of the AUT that are covered at least once by at least a test case of the test suite T:

$$\eta(T) = 100 * \frac{|\bigcup_{t \in T} Cov(t)|}{|LOC|}$$

3.1 Subjects

Five real-world open source Android Applications have been selected for our study; they are all published and freely available on the Google Play market. Some details about these application are reported in Table 2. They are all medium sized applications, with a number of LOCs varying from 2308 lines (AUT1) to 6770 lines (AUT3).

Application preconditions can affect the effectiveness of the generated test cases [5]. For the purpose of this experimentation, we chosen the same set of preconditions for each application and used it in all the experiments with both the techniques (as an example, for AUT1 we preloaded the same dictionary in the SD card before the execution of each test case).

3.2 Experiment Environment and Setup

The experimentation has been carried out by using two tools that we have implemented, i.e. the Android Ripper tool and the AGRippin tool.

The Android Ripper tool [2] [5] has been used to systematically explore the GUIs of Android applications with a breadth-first strategy. Each branch of the exploration carried out by the Android Ripper tool is terminated when a GUI interface is found that is equivalent (in the sense that it has the same widgets and event handlers) to a previously visited one. The Android Ripper tool produces a test suite composed of test cases corresponding to the explored execution paths.

The Android Ripper tool is composed of two main components. The *Driver* component is responsible for the execution of the exploration algorithm and for the generation of the resulting test cases in form of Android JUnit test cases exploiting the Robotium library [3]. The *Device* component is deployed and executed in the context of an Android emulator and is able to execute actions on the AUT, to extract the obtained GUI interfaces and to send their description to the *Driver* component via the Android Debug Bridge (ADB)[4]

[2]https://github.com/reverse-unina/AndroidRipper
[3]https://code.google.com/p/robotium/
[4]http://developer.android.com/tools/help/adb.html

Table 2: Android Applications (AUTs)

	Application	Description	Link	LOCs	Activities
AUT1	AardDict 1.4.1	A dictionary application	https://github.com/aarddict/android	2308	7
AUT2	TomDroid 0.7.1	A manager for notes	https://code.launchpad.net/tomdroid	4167	10
AUT3	OmniDroid 0.2.1	A manager for device automated tasks and actions	https://code.google.com/p/omnidroid/	6770	16
AUT4	AlarmClock 1.7	An alarm clock	https://code.google.com/p/kraigsandroid/	2320	5
AUT5	BookWorm 1.0.18	A manager for book collections	https://code.google.com/p/and-bookworm/	3190	10

utility. The code coverage has been measured by means of the Emma utility [5] included in the Android SDK.

The AGRippin tool[6] has been realized on top of the Android Ripper tool by implementing an *AGR* component responsible of the execution of the proposed technique. The *AGR* component interacts with both the components of the Android Ripper tool.

The experimentation has been carried out on 6 different Intel I5 PCs with a clock frequency of 3.0GHz, 4GB of RAM and Windows 7 64bit operative system. On these machines we installed an Android Virtual Device [7] (AVD) emulating the Android Gingerbread 2.3.3 operative system, with 512MB of RAM and an emulated 64MB SD Card.

We started the experimentation by carrying out an exploration of each AUT by means of the Android Ripper tool that generated a test suite. We considered these test suites as a result of an Hill Climbing exploration because the Ripper strategy consists of the selection, at each step, of the most promising action, i.e. of an action that has not been previously executed. The test suite generated by the Android Ripper tool has been used, too, as the initial solution of the search based testing technique.

The AGRippin technique has been configured in our experimentation by fixing the parameters shown in Table 3. The *Crossover ratio* and the *Mutation ratio* respectively represent the fraction of the test cases of a test suite that are involved in a crossover or in a mutation at a given iteration. In accordance with the suggestions of Mitchell et al. [19] we set a higher value for the Crossover ratio with respect to the Mutation ratio. The *Number of Iterations* represents the termination condition of our algorithm in terms of the number of performed iterations. We fixed an arbitrary value of 30 in this experimentation. In order to take into account the randomness of our search based testing technique, we executed the AGRippin technique six times with six different seeds for any AUT.

Table 3: Configuration Parameters Values

Parameter	Value
Crossover ratio	20%
Mutation ratio	5%
Number of Iterations	30

3.3 Results and Discussions

Table 4 reports the results obtained by the execution of our experimentation on the five AUTs.

The first column of the table reports the effectiveness $\eta(T_0)$ of the test suite T_0 generated by the Hill Climbing (HC) technique implemented by the Android Ripper tool. The columns labeled $\mu(\eta(T))$ and $\sigma(\eta(T))$ respectively report the average and the standard deviation of the effectiveness η of the test suites T generated by the AGRippin technique (abbreviated in AGR in Table 4) in six different executions featuring different random seeds. The fourth column of the table reports the maximum number of interfaces discovered by AGRippin that have not been discovered by HC. The fifth column reports the number of test cases composing both the test suites generated by HC and AGRippin. Finally, the last two columns respectively report the HC execution time and the average execution time of AGRippin (after 30 iterations), measured on the same machines. The results in this table show that for all the considered AUTs the AGRippin technique is able to provide an increase in coverage with respect to the Hill Climbing technique that varies from 1% (for AUT4) to 24% (for AUT1), so we can conclude that the proposed RQ has a positive answer. As regards the execution time, we can observe that the average time needed to execute 30 iterations with AGRippin varies from 6 to 12 times the amount of time needed to execute HC. The values of standard deviation $\sigma(T)$ show that the effectiveness of AGRippin depends on the randomness in a remarkable way. We can hypothesize that the execution of a larger number of parallel sessions can provide improvements in the effectiveness of the AGRippin technique without increasing the execution time.

In order to show an example of the dependence of the effectiveness on the number of iterations, Figure 4 reports the effectiveness trends observed for the Bookworm application (AUT5). The figure shows the trends of the coverage of the test suites obtained by six different executions (with six different random seeds) of the AGRippin technique (named AGR1, AGR2, AGR3, AGR4, AGR5, AGR6 in figure) as the number of iterations increases. The dashed line shown in figure represents the coverage percentage provided by the test suite generated by the HC technique. We can observe that each execution of AGRippin constantly reaches higher values of effectiveness than HC. In the Bookworm application, one interface has been discovered by each AGRippin execution and in these cases a large portion of unexecuted code have been systematically been explored by the Hill Climbing technique. We can note its effect by observing the rapid rising in the effectiveness that occurs once for each AGRippin execution. The values of effectiveness after 30 iterations are slightly different between them. On the basis of this phenomenon we can hypothesize that the execution of a larger number of iterations may provide better results and a reduced dependency on randomness. Similar considerations can be done for all the other AUTs.

In order to provide more details about the capability of the AGRippin technique to cover unexplored portions of code, we examined in details the differences in coverage between the executions of the HC and AGRippin techniques. We recognized improvements in coverage due to the Crossover

[5] http://emma.sourceforge.net/

[6] https://github.com/reverse-unina/agrippin

[7] https://developer.android.com/tools/devices/index.html

Table 4: Experimental Results

	HC	AGR				Execution Time (hours)	
	$\eta(T_0)$	$\mu(\eta(T))$	$\sigma(\eta(T))$	**Discovered Interfaces**	$\lvert T \rvert$	**HC**	**AGR**
AUT1	43.07%	67.10%	0.26%	1	51	3.5	22
AUT2	28.08%	32.61%	2.45%	0	51	2.5	30
AUT3	51.58%	58.31%	2.28%	0	162	6	55
AUT4	66.90%	68.00%	1.21%	0	68	2.8	20
AUT5	40.34%	47.22%	0.45%	1	50	3.2	23

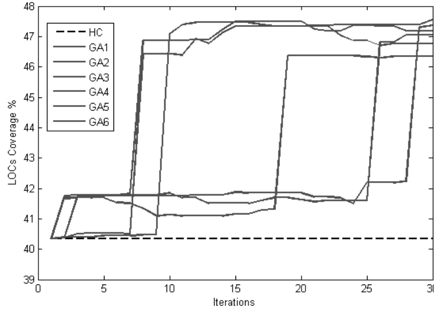

Figure 4: Effectiveness Trends for AUT5

operator, to the Mutation operator and to the Combination technique.

As regards the improvements due to the crossover operator, we observed that in some cases the mixing of two different test cases produced new execution sequences that are able to show different behaviors of the AUT. As an example, for AUT2 the crossover operator generates a new test case executing the backup functionality after the changing of its settings causing the execution of a different backup scenario. Another example of unexplored code executed by a crossover operator is, in AUT5, the one related to the sequential execution of the insertion of a book in the book list followed by the visualization of this book list. In the test suite generated by HC, the visualization was executed only with an empty book list whereas AGRippin were able to visualize book lists that are not empty.

As regards the improvements in effectiveness due to the mutation operator, we report two exemplar cases. An AUT1 functionality is the search in a vocabulary of a string included in an input text field. Whereas the HC technique only provided a random text to this input field, the mutation operator implemented in AGRippin generated a new test case with an input text value belonging to the equivalence class of the English words. This mutation caused the execution of a portion of code related to the retrieval of the word in the dictionary and to the visualization of one undiscovered interface showing the list including one or more search results. Another example is the one found in AUT4, where the insertion of an input value belonging to the negative numbers equivalence class in a specific text field caused the execution of a portion of code executing a validating check that has not been tested by HC.

Finally, the combination technique has been applied in two cases to explore two interfaces discovered in AUT1 and AUT5 (corresponding to two of the cases described above). In these cases the application of the HC technique on these interfaces caused a great improvement in code coverage (about 5% of improvement in both the cases).

4. CONCLUSIONS AND FUTURE WORKS

In this paper a novel search based testing technique has been proposed and implemented in the context of Android applications. We evaluated its effectiveness by carrying out a case study involving five open source Android applications. We demonstrated that the technique is more effective than an Hill Climbing technique based on the systematic exploration of the GUI events executable on an Android application, in terms of source code coverage.

In order to generalize the promising results shown by this preliminary experimentation, in future we will extend our experimentation to a larger set of Android applications and we will extend our comparative analysis to random techniques and other techniques for automatic test case generation proposed in literature.

One of the objectives of the future experimentation will be the tuning of the algorithm parameters values (such as the crossover ratio, the mutation ratio, the number of iterations and the test suite size) and the evaluation of their influence on the effectiveness of the generated test suites and on the number of iterations needed to reach this level of effectiveness. As regards the crossover and mutation ratio, we plan to implement a technique based on adaptive variations (as the one proposed by Srinivas and Ptnaik in [21]) in order to reduce the probability that the generated test suites maintain the same coverage for many consecutive iterations, as experienced in some of the case studies. As regards the number of iterations we will carry out longer experiments in order to evaluate if some phenomena of convergence of the coverage to a global maximum may be observed. Finally, we plan to test a variant of the technique in order to increase the test suite size when new test cases are generated by the Hill Climbing technique in order to avoid the loss of important test cases due to the selection operator.

5. ACKNOWLEDGMENTS

The authors would like to thank Carmine Cirillo for his contribution to the implementation of some components of the Agrippin tool.

6. REFERENCES

[1] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 51(6):957–976, June 2009.

[2] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, Nov 2010.

[3] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. Memon. Exploiting the saturation effect in automatic random testing of android applications. In *The Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2015)*, 2015.

[4] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon. Mobiguitar – a tool for automated model-based testing of mobile apps. *Software, IEEE*, PP(99):1–1, 2014.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[6] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *SIGPLAN Not.*, 48(10):641–660, Oct. 2013.

[7] J. Baker. Adaptive selection methods for genetic algorithms. volume Proceedings of the First International Conference on Genetic Algorithms and Their Applications. Erlbaum, 1985.

[8] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 2013.

[9] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Software, IEE Proceedings -*, 150(3):161–175, June 2003.

[10] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[11] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[12] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, New York, NY, USA, 2013. ACM.

[13] M. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 15–24, Oct 2013.

[14] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10, April 2015.

[15] C.-H. Liu, C.-Y. Lu, S.-J. Cheng, K.-Y. Chang, Y.-C. Hsiao, and W.-M. Chu. Capture-replay testing for android applications. In *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, pages 1129–1132, June 2014.

[16] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[17] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[18] H. Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In R. Männer and B. Manderick, editors, *PPSN*, pages 15–26. Elsevier, 1992.

[19] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254. MIT Press, 1991.

[20] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 29–35, June 2012.

[21] M. Srinivas and L. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):656–667, Apr 1994.

[22] G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In G. J. Rawlins, editor, *Foundations of genetic algorithms*, pages 94–101. Morgan Kaufmann, San Mateo, CA, 1991.

[23] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 377–386, Washington, DC, USA, 2011. IEEE Computer Society.

[24] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. User guided automation for testing mobile apps. In *23rd IEEE International Conference on Program Comprehension, (ICPC'15)*, volume 1, page to appear, May 2015.

[25] D. Whitley and K. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, 1988.

[26] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.