A Technique for Parallel GUI Testing of Android Applications

Porfirio Tramontana, Nicola Amatucci, Anna Rita Fasolino

A Technique for Parallel GUI Testing of Android Applications

Porfirio Tramontana[1], Nicola Amatucci[1], and Anna Rita Fasolino[1]

University of Naples "Federico II", Napoli, Italy
{ptramont,nicola.amatucci,fasolino}@unina.it

Abstract. There is a large need for effective and efficient testing processes and tools for mobile applications, due to their continuous evolution and to the sensitivity of their users to failures. Industries and researchers focus their effort to the realization of effective fully automatic testing techniques for mobile applications. Many of the proposed testing techniques lack in efficiency because their algorithms cannot be executed in parallel. In particular, Active Learning testing techniques usually relay on sequential algorithms.
In this paper we propose a Active Learning technique for the fully automatic exploration and testing of Android applications, that parallelizes and improves a general algorithm proposed in the literature. The novel parallel algorithm has been implemented in the context of a prototype tool exploiting a component-based architecture, and has been experimentally evaluated on 3 open source Android applications by varying different deployment configurations.
The measured results have shown the feasibility of the proposed technique and an average saving in testing time between 33% (deploying two testing resources) and about 80% (deploying 12 testing resources).

1 Introduction

Nowadays, the diffusion of mobile applications is continuously increasing, and these applications are often characterized by a very tight time-to-market to follow the evolution of the users needs and trends. On the other hand, this rapid development process can produce mobile applications containing a large number of faults. Crashes and other failures originated by these faults may compromise the quality of the app and can be responsible of drastic losses in terms of users that will soon uninstall an unreliable application[1]. In the last years academia and industry have shown a great interest for fully automatic testing techniques and tools as shown by the large number of proposals that can be found in literature [30,25].

In the context of Android applications, most of these techniques can be classified in three distinct categories: Random testing techniques, Model Based testing techniques and Active Learning testing techniques [1].

[1] https://blog.helpshift.com/80-users-delete-mobile-apps/

Random Testing Techniques are fully automatic testing techniques that choose user or system events at random and execute them on the application under test without taking into account the previously executed events. For example, the Monkey tool[2] is a command line executable testing tool included 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 proposed in [13,17,20,26]. Random testing techniques can be applied without any previous knowledge of the application under test and can be easily parallelized by executing different independent testing sessions on different testing resources. Although these techniques have demonstrated their effectiveness in many contexts, a well known limitation is due to the possible redundancy of the executed test cases, that causes a remarkable inefficiency. For this reason, such techniques often require long running times before achieving significant effectiveness values [9,1]. In addition, the trend of the discovery of unexplored application behaviors tends to drastically decrease with the increase of the running time, so it is difficult to choose a termination criterion that ensures an optimal trade-off between effectiveness and efficiency [3,26].

More efficient solutions to these testing problems are represented by Model Based and Active Learning testing techniques. Model Based techniques are able to generate and execute test cases from structural and/or behavioral models of the application. Examples of models used for the automatic generation and execution of test cases are Finite State Machines [22,19,23,29], Sequence Diagrams [5], Activity Diagrams [11,15], GUI Trees [27,28,3] and Labeled State Transition Machines [24]. A limitation of these techniques is represented by the difficulty to design or reverse engineer an accurate model describing the structure and the behavior of the application under test.

A specialization of Model Based techniques is represented by Active Learning testing techniques [8]. These techniques (also called Model Learning techniques) do not need any existing model of the application under test but they are able to dynamically build models during an automatic exploration process. The exploration itself is driven by the dynamically built model. Amalfitano et al. [1] have studied the similarities between different Active Learning approaches presented in literature in the context of GUI testing of Android applications. They have found that all the approaches consist of iterative algorithms in which, at each iteration, an user or a system event that is executable on the current GUI of the application under test is scheduled and executed, and the resulting GUI is used to refine the model. Active Learning techniques usually terminate their execution when all the relevant events executable on the GUIs of the application under test have been executed at least once. Many Active Learning techniques and tools supporting GUI testing of Android applications have been proposed in the last years both by academy and industry [25], including for example Android Ripper [4], A^3E [6], SwiftHand [8], Dynodroid [17] and Sapienz [20]. Google has released and made available to developers an Active Learning testing tool in the

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

form of a cloud service called Android Robo Test[3], that represents a simple but not very effective tool.

Active Learning techniques are generally difficult to be parallelized since they are based on iterative algorithms where the selection of the next event to be tested is based on the analysis of the reconstructed model of the application, that is refined at each iteration. The need for a centralized storing and management of this model introduces a constraint for the parallelization of the testing activities.

This paper presents an evolution of an instance of the Unified Online Testing algorithm presented in [1]. This evolution allows the parallel execution of some portions of the algorithm, with a consequent saving in terms of testing time with respect to its sequential version. This technique has been implemented by a prototype tool extending the Android Ripper tool [4,2]. We carried out a preliminary experimentation of the tool by executing it on 3 different Android applications by varying the number of testing machines and the number of virtual devices instances for each machine, in order to assess the savings in testing time with respect to the non-parallel configuration.

The paper is organized as follows: in Section 2 the most relevant parallel Active Learning testing techniques found in the scientific literature are presented while in Section 3 our proposed algorithm is presented. The characteristics of the prototype tool implementing the algorithm are reported in Section 4. The results of some experiments carried out on 3 open source Android applications are presented in Section 5, together with a discussion of the obtained results and of the directions for the future improvements of the technique and of the tool. Finally, conclusions and future works are reported in Section 6.

2 Related Work

As highlighted in [25], only few automatic techniques and tools supporting GUI testing of Android applications have been designed having parallel execution in mind.

Neamtiu et al. [14] in 2011 presented a first technique based on the parallel execution of the Monkey tool to generate random or deterministic event sequences, while Mahmood et al. [18] in 2014 proposed EvoDroid, a tool that exploits parallel execution in the context of a cloud platform to increase the efficiency of the Search-based technique they implemented. Similarly Mao et al. [20] exploited parallel execution of test cases to reduce the execution time of their multi-objective search based testing strategy implemented by the tool Sapienz.

As regards Parallel Active Learning testing techniques, the first relevant contribution in the literature is the one provided by Meng et al. [21], that presented ATT, a master-slave testing framework supporting the parallel and distributed execution of test cases on Android applications. With ATT it is possible to distinguish an AgentManager component that acts as master and interacts with the slave nodes, called Agents. Each Agent is in charge of interacting with one

[3] https://firebase.google.com/docs/test-lab/robo-ux-test

Android Virtual or Real Device: it receives information about the GUI and interacts with the target app through a service running on the target device. ATT has been applied to different testing processes involving the re-execution of test cases generated by the Capture and Replay tool RERAN [10], the random testing tool called Monkey+ and the hybrid approach called UGA [16].

Wen et al. [28] proposed in 2015 a framework called PATS aiming at the reduction of the execution time of an Active Learning testing process by using a master-slave model. The dynamic exploration algorithm performed by the testing process is carried out by a set of slave nodes that analyze a GUI, dynamically elicit event sequences to be fired and actually fire events on the application under test. These events are managed by a coordinator that guides the process dispatching new GUIs to the slave nodes; this component is also in charge for avoiding redundancies within the model.

More recently, Cao et al. [7] have presented ParaAim, a tool for the systematic exploration of Android apps designed for parallel execution. It explores the GUI states of an app using the exploration technique presented in [12]. When a newly encountered Activity of the app is discovered for the first time, it starts a new task considering this state as a starting point. The proposed approach is based on the master-slave pattern where the master owns a queue of tasks that are scheduled on independent slave nodes; each slave node restores the GUI state by replying the event sequence leading to it, then it continues the exploration from there. Moreover, to speed up the exploration, there are a prioritization algorithm and an event sequence minimization heuristic to reduce the costs related to replay the event sequences. The experiments reported in [7] show how the exploration speed increases almost linearly as the number of devices increases. ParaAim, similarly to our approach, adopts a master-slave architecture and distributes tasks on the slave nodes but it also introduces a prioritization strategy and performs an event sequence minimization task. A weakness of ParaAim with respect to our tool is related to the granularity of the analysis: ParaAim is guided by Activity coverage, whereas our tool is driven by the more fine grained coverage of executable events [1].

3 A Parallel Algorithm for the Automatic Exploration of Android Applications

In this section the parallel Active Learning algorithm that we have designed for the automatic exploration and testing of Android applications will be presented. This algorithm has been derived from a sequential Active Learning Testing algorithm that implements an iterative exploration of an application under test (AUT). The pseudo-code of this sequential algorithm is shown in Algorithm 1. This algorithm has been obtained as an instance of the more general Unified Online Testing Algorithm presented in [1].

The algorithm begins by executing the *initializeAppModel* operation that starts the AUT and initializes the model *appModel* with the description of the first GUI of the AUT. The GUI description includes a subset of its widgets and

Algorithm 1 Sequential Active Learning Testing algorithm

```
1: appModel ← initializeAppModel();
2: stopCondition ← evaluateStopCondition();
3: while (!stopCondition) do
4:     fireableEvents[] ← extractEvents(appModel)
5:     eventsSequence ← scheduleEvents(fireableEvents[]);
6:     CurrentGUIDescription ← runEvents(eventsSequence);
7:     appModel ← refineAppModel(CurrentGUIDescription);
8:     stopCondition ← evaluateStopCondition();
9: end while
```

the values of a subset of its attributes (e.g. buttons and text fields and attributes such as id and event handlers). At each iteration, the algorithm analyzes the description of the current GUI interface in the *appModel* and evaluates if there are new events that can be fired on it. The set of considered events is limited to the ones for which an explicit event handler has been implemented in the AUT code. The *extractEvents* operation pushes this set of executable events (and the corresponding sequences of events preceding each of them) on the *fireableEvents* queue. Successively, the *scheduleEvents* operation pops an executable events sequence from the *fireableEvents* queue and executes it on the AUT (*runEvents* method). The *refineAppModel* adds the description of the GUI obtained after the execution of the sequence of events to the *appModel*. The algorithm continues until there are no more events to be scheduled, i.e. until the *fireableEvents* queue is empty.

In order to design a parallel version of this algorithm, some observations have to be taken into account. Most of the operations performed by this algorithm involve the analysis or the update of the *appModel*: (1) the *initializeAppModel* operation inserts into the *appModel* the description of the starting GUI of the AUT; (2) the *extractEvents* method extracts from the model the set of events that can be executed on the current GUI of the AUT; (3) the *refineAppModel* operation updates the *appModel* with the description of the last explored GUI. All these operations cannot be parallelized since they need the exclusive access to the *appModel* data structure.

The *scheduleEvents* and the *runEvents* methods, instead, can be concurrently executed. In particular, a Degree of Parallelism P can be achieved for these operations if P testing resources (i.e. virtual or real devices) are available. In this case, the semantic of the *scheduleEvents* method is modified in order to select a set of P sequences of events. The *runEvents* method can be transformed in the *runParallelEvents* one, that sends each of the P scheduled sequences of events to one of the P available resources. The Algorithm 1 is consequently transformed in the Parallel Active Learning testing algorithm (Algorithm 2).

In details, the *scheduleParallelEvents* method returns a set of P sequences of events *eventsSequence[]* that is passed to the *runParallelEvents* method. This method distributes the P sequences of events to the P different testing resources. The *refineAppModel* operation sequentially updates the *appModel* taking into

Algorithm 2 Parallel Active Learning Testing Algorithm

```
1: appModel ← initializeAppModel();
2: stopCondition ← evaluateStopCondition();
3: while (!stopCondition) do
4:     fireableEvents[] ← extractEvents(appModel)
5:     eventsSequence[] ← scheduleParallelEvents(P, fireableEvents[]);
6:     CurrentGUIDescription[] ← runParallelEvents(P, eventsSequence[]);
7:     appModel ← refineAppModel(CurrentGUIDescription[]);
8:     stopCondition ← evaluateStopCondition();
9: end while
```

account all the P GUI instances received from the testing resources. Of course, if there are less than P items in the *fireableEvents* queue, only a subset of the resources will be used in that iteration of the algorithm.

4 Architecture and Implementation of the Tool

The Algorithm 2 presented in the previous section has been implemented in a prototype called *Parallel Android Ripper*, that is based on the Android Ripper[4] tool presented in [1]. The system is built according to a component based architecture and can be distributed on different machines. The architecture includes the following two types of components:

- The *Coordinator* component that is responsible of the execution of the Algorithm 2 and that has exclusive access to the *appModel* data structure; it coordinates the execution of the whole process.
- The *Test Case Executor* components that receive from the *Coordinator* sequences of events to be executed and inject them into the Android Virtual or Real Device they manage.

The abstract architecture of the Parallel Android Ripper can be deployed in different ways by changing the number of involved machines, the number of *Test Case Executor* components deployed on each of these machines and the type of testing resources (i.e. Android Virtual or Real Devices).

For example, Figure 1 shows a possible deployment scenario where the *Coordinator* component is deployed on a physical machine storing the *appModel*, too, and there are three machines each one hosting two instances of the *Test Case Executor* component. Each *Test Case Executor* instance is connected to an Android Virtual Device (AVD) deployed on the same machine, and is able to install the AUT and run a test case corresponding to the scheduled sequence of events. The Degree of Parallelism P of this deployment scenario is equal to 6 since there are 6 different *Test Case Executor* components that can run in parallel.

[4] https://github.com/reverse-unina/AndroidRipper

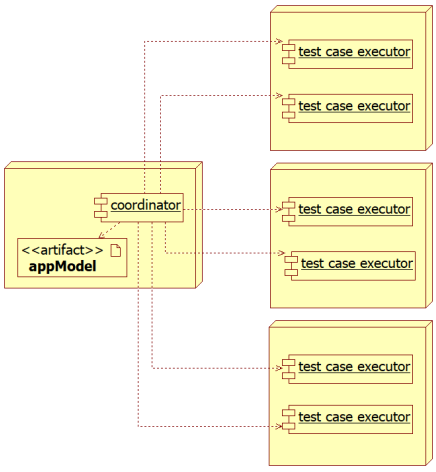

Fig. 1. Deployment of the Parallel Android Ripper with 4 distinct machines and 6 *Test Case Executor* components

A prototype of the Parallel Android Ripper tool has been implemented in order to evaluate its feasibility and to assess its performance in the context of the experiments presented in the following section.

The *Coordinator* component has been deployed on an Application Server and includes a Web Application and a REST Web Service. The Web Application allows the user (1) to upload the apk of the AUT, (2) to select the set of testing machines hosting the instances of the *Test Case Executor* components and the number of *Test Case Executor* components instantiated on each machine, (3) to start and control the execution of the whole process. The REST Web Service is directly invoked by the Web Application and is responsible for the execution of the Active Learning algorithm and for the communication with the *Test Case Executor* components. The *Coordinator* component returns as outputs the reconstructed GUI Tree Model of the AUT, the code coverage achieved by the exploration and the log files reporting the encountered failures and exceptions.

Each *Test Case Executor* component exposes an RMI interface featuring the methods needed to drive the execution of a sequence of events on an AVD and to return a description of the current GUI of the AUT. In detail, each *Test Case Executor* component generates a testing package including a JUnit test case featuring the scheduled sequence of events and the *AUT*, then it installs and starts the test case on an AVD via the Android Debug Bridge (ADB) tool. The description of the GUI obtained after the execution of this sequence of events is returned to the *Coordinator* component at the end of the execution of the test case.

5 Experimental Evaluation

This Section reports the results of an experimental evaluation of the performance of the Parallel Android Ripper tool involving three open source Android applications and different deployment scenarios having different Degrees of Parallelism P. In particular, the main objectives of the experiments are the following:

- to evaluate if and how the effectiveness of the Parallel Android Ripper tool (measured in terms of the code coverage) is influenced by the degree of parallelism P;
- to evaluate how much testing time can be saved by deploying multiple testing machines and multiple *Test Case Executor* instances on each testing machine.

5.1 Variables and Metrics

The independent variable of the experiment presented in this section is represented by the deployment configuration of the tool. Different deployments have been considered by varying the following two parameters:

n_m that is the total number of different physical machines hosting the *Test Case Executor* components;
n_e that is the number of *Test Case Executor* components deployed for each of the n_m machines.

As a consequence, the degree of parallelism P can be defined as the product between the number of physical machines and the number of *Test Case Executor* components deployed for each machine:

$$P = n_m * n_e$$

We have considered 6 different deployment configurations having 1, 2 or 6 physical machines and 1 or 2 instances of the *Test Case Executor* per machine, with resulting degrees of parallelism P between 1 and 12. The following 6 configurations have been considered:

- $C_{1,1}$: ($n_m = 1$, $n_e = 1$, $P = 1$)
- $C_{1,2}$: ($n_m = 1$, $n_e = 2$, $P = 2$)
- $C_{2,1}$: ($n_m = 2$, $n_e = 1$, $P = 2$)
- $C_{2,2}$: ($n_m = 2$, $n_e = 2$, $P = 4$)
- $C_{6,1}$: ($n_m = 6$, $n_e = 1$, $P = 6$)
- $C_{6,2}$: ($n_m = 6$, $n_e = 2$, $P = 12$)

The first configuration $C_{1,1}$ implements a sequential version of the algorithm, with a single *Coordinator*, a single *Test Case Executor* and a single AVD. This configuration has been considered as a benchmark for the evaluation of the savings in testing time obtained by varying the degree of parallelism P between 2 and 12.

The dependent variables measured for each execution of the testing tool are the testing time tt needed to complete the process and the code coverage percentage cc. We have considered the code coverage instead of the number of observed failures or crashes because it provides a more detailed view of the coverage of the application behaviors.

In order to evaluate the saving of testing time due to the selection of different deployment configurations having different degrees of parallelism P, we have introduced a metric called Speed Improvement factor S. The Speed Improvement among two configurations C_i and C_j is defined by the following expression:

$$S(C_i, C_j) = \frac{tt(C_j) - tt(C_i)}{tt(C_j)} = 1 - \frac{tt(C_i)}{tt(C_j)} \quad (1)$$

For example, if we want to measure the Speed Improvement of the configuration $C_i = C_{2,1}$ with respect to the reference configuration $C_j = C_{1,1}$, and the testing time using $C_{1,1}$ is $tt(C_{1,1}) = 34$ minutes, while the testing time using $C_{2,1}$ is only $tt(C_{2,1}) = 23$ minutes, then the Speed Improvement factor is $1 - (23 \div 34) = 0.32$. In other words, 32% of the testing time can be saved by deploying two testing machines each one hosting a single *Test Case Executor* component, instead than using a single machine.

5.2 Experimental Objects

The objects of the experiments are 3 small-sized open source Android applications that have been selected from the Google Play market and for which the source code is online available. Table 1 provides some pieces of information about the selected applications including a short description, the number of classes, the number of Activity classes, the number of methods, the number of event handlers methods and the number of LOCs. These applications have been already considered as case studies in the experiments carried out in [1].

Table 1. Characteristics of the AUTs involved in the experiments

ID	Application	Version	Description	# Classes	# Activ.	# Methods	# Event handlers	# LOCs
AUT1	TicTacToe	1.0	Simple game	13	1	47	16	493
AUT2	TippyTipper	1.2	Tip calculator app	42	6	225	70	999
AUT3	Tomdroid	0.7.1	Note-taking app	133	10	707	117	3860

5.3 Experimental Setup and Procedure

The tool has been deployed according to the architecture presented in section 4 based on a single *Coordinator* component and a set of *Test Case Executor* components deployed on a set of different testing machines and driving AVDs running on the same machines.

More in details, the *Coordinator* component has been deployed on a PC running the Windows 7 Operating System, featuring an Intel I5 3.0GHz processor and 8GB of RAM; each *Test Case Executor* component has been deployed on PCs having the same characteristics. Each *Test Case Executor* manages the execution of the test cases on a different Android Virtual Device (AVD) running on the same machine; the AVD is configured to emulate an Android device having 2 GB of RAM, a 512 MB SD Card, and running the Android Nougat (7.0) operating system. All the machines are deployed in the context of the same network infrastructure, featuring a static network configuration.

Each AUT has been tested for each of the six configurations $C_{1,1}$, $C_{1,2}$, $C_{2,1}$, $C_{2,2}$, $C_{6,1}$, $C_{6,1}$. For each run the testing time needed to complete the execution of the testing process and the achieved code coverage percentage have been measured. The code coverage has been measured exploiting the Emma code coverage tool[5] included in the standard distribution of the Android development kit. The Speed Improvement factor has been evaluated for each possible pair of different configurations. No specific preconditions have been set for each of the 3 AUTs; moreover, each AUT has been re-installed after the execution of each test case. A delay of two seconds has been added before each execution of two consecutive events.

5.4 Results

Table 2 reports the testing time *tt* (in minutes) measured for each of the 3 AUTs and for each of the 6 considered configurations. In addition, the last column reports the corresponding code coverage *cc*, measured in percentage. In the last row, the Degree of Parallelism P of each configuration is reported, too.

Table 2. Measured Results for each Configuration and each AUT

	tt (minutes)						
	$C_{1,1}$	$C_{1,2}$	$C_{2,1}$	$C_{2,2}$	$C_{6,1}$	$C_{6,2}$	cc %
AUT1	34	23	19	14	12	7	78%
AUT2	85	62	59	44	26	19	75%
AUT3	164	113	107	74	50	32	36%
P	1	2	2	4	6	12	

The data reported in Table 2 show that the code coverage percentage does not depend on the deployment configuration, thus the introduction of the parallelism in the sequential algorithm does not produce any variation in its effectiveness. By observing the data from left to right, it is clear that the testing time is strongly affected by the degree of parallelism P and it decreases as P increases.

More in detail, we have evaluated the Speed Improvement that can be obtained (1) by increasing the number of machines and (2) by increasing the number of *Test Case Executor* components. Table 3 reports the values of the Speed

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

Improvement factor (measured in percentage) evaluated by comparing deployment configurations having different numbers of machines and the same number of *Test Case Executor* components per machine.

Table 3. Values of the Speed Improvement factor evaluated among configurations having the same number of *Test Case Executor* components per machine

	$S(C_{2,1}, C_{1,1})$	$S(C_{2,2}, C_{1,2})$	$S(C_{6,1}, C_{1,1})$	$S(C_{6,2}, C_{1,2})$
AUT1	42%	38%	63%	69%
AUT2	31%	29%	69%	69%
AUT3	35%	35%	70%	72%
Average	36%	34%	66%	70%

By analyzing the second and the third column of Table 3 we can observe that if the number of machines is doubled (from 1 to 2), there is a speed improvement between 29% and 42% for the three considered AUTs (35% in average). Analogously, the last two columns show that if the number of machines is increased of 6 times (from 1 to 6), the speed improvement varies between 63% and 72% (68% in average).

Table 4 reports the values of the Speed Improvement factor (measured in percentage) evaluated by comparing deployment configurations having the same number of machines and, respectively, 1 or 2 *Test Case Executor* components per machine. The table shows that doubling the number of *Test Case Executor*

Table 4. Values of the Speed Improvement factor measured between configurations having the same number of machines and different numbers of *Test Case Executor* components per machine

	$S_i(C_{1,2}, C_{1,1})$	$S_i(C_{2,2}, C_{2,1})$	$S_i(C_{6,2}, C_{6,1})$
AUT1	32%	27%	43%
AUT2	27%	26%	26%
AUT3	31%	31%	36%
Average	30%	28%	35%

components per machine, the speed of the testing process is increased in average only of 31% against the average of 36% of improvement obtained by doubling the number of testing machines. So, doubling the testing machines is preferable to doubling the testing resources deployed on the same machines.

Figure 2 shows that the increase in Speed Improvement is always less than linear with respect to the corresponding increase of the degree of parallelism P. In detail, the continuous line in the figure represents the ideal trend of the Speed Improvement evaluated with respect to the configuration $C_{1,1}$ and the points represent the measured values of S with respect to the $C_{1,1}$ configuration. Figure 2 shows that all the points are below the line, in particular the ones

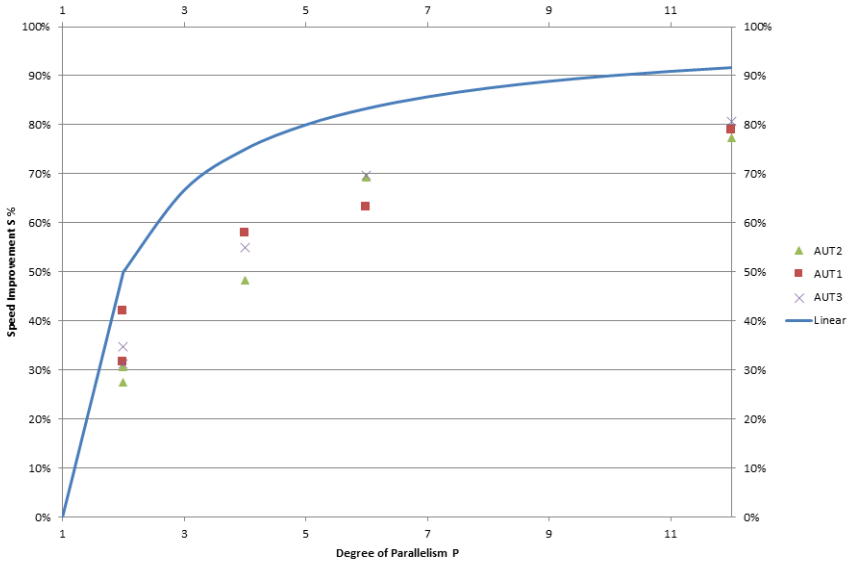

Fig. 2. Trend of the Speed Improvement with respect to the configuration $C_{1,1}$ for different configurations having different degrees of parallelism P

corresponding to configurations having two instances of the *Test Case Executor* for machine.

These data can be compared with the ones reported in [28] according to which the PATS tool provides in average a Speed Improvement S of 27% on a configuration having two different test executors deployed on two different machines instead that only one, whereas our tool has provided in average a Speed Improvement of 35% with two *Test Case Executor* components deployed on two different machines and of 31% when deployed on the same machine. The ParaAim tool [7] obtains slightly better results in time saving when deploying two or four devices, with a corresponding increase in testing effectiveness, too. However, the testing effectiveness obtained by ParaAim is measured in terms of Activity coverage and is quite small since it does not exceed 15% on average. Although these data are related to different sets of applications (3 AUTs for our tool, 4 AUTs for PATS, 20 AUTs for ParaAim) and different coverage metrics, it appears that the performance of our tool are encouraging.

In conclusion, this experimental evaluation has shown that the introduction of a parallel algorithm does not influence the effectiveness of the testing tool in terms of achieved code coverage, whereas it provides a reduction of the testing

time with respect to the sequential algorithm. More in detail, a larger speed improvement can be obtained by increasing the number of testing machines, whereas an increase of the number of the *Test Case Executor* components deployed on the same testing machine provides minor improvements.

5.5 Discussions and Possible Improvements

In this section a discussion of the results obtained by the analysis of the experimental results will be presented, with the aim of proposing ideas for further improvements of the efficiency of the proposed algorithm and of its prototype implementation.

Four limiting factors affecting the performance of the proposed tool have been recognized: (1) the presence in the algorithm of methods that are not parallelized at all; (2) the losses in efficiency occurring when the number of testing resources grows, (3) the delay due to the waiting for the termination of the different scheduled event sequences, (4) the concurrency between different *Test Case Executor* components deployed on the same machine. In the following, we will discuss each of these factors and will indicate some proposals to reduce their influence on the performance of the tool.

Parallelization of the algorithm. The unique operation of the Algorithm 2 that has been parallelized is the *RunParallelEvents* operation. We have primarily focused our attention on the parallelization of this operation since it is usually the most costly one in terms of testing time.

Other methods of the algorithm could be partially parallelized. For example, by adopting the *Breadth First* scheduling strategy, the *extractEvents* operation extracts from the *appModel* some events executable on the last explored GUIs and pushes them on the bottom of the *fireableEvents* queue, while the *scheduleParallelEvents* operation selects the first P events on top of the same queue. When the queue contains more than P events, the two operations can be executed concurrently. The restart time needed to uninstall and reinstall the AUT can be reduced by performing these operations in parallel just after the returning of the GUI description to the coordinator.

Reduction of the speed improvement for growing degrees of parallelism. The experiments have shown that an increase of the degree of parallelism P progressively reduces the testing time needed to execute the proposed algorithm. Anyway, it is possible to observe that, when the degree of parallelism P grows, there are iterations of the algorithm for which there are less than P fireable events, so that not all the testing resources can be used in parallel. In order to have an idea of this phenomenon, we measured the number of iterations of the Algorithm 2 executed on $AUT1$ for different values of P. Table 5 shows the number of iterations needed to complete the testing algorithm and, in the last row, the minimum number of iterations needed if all the testing resources are used at each iteration (i.e. the ratio between the total number of executed events and the the degree of parallelism). We can observe that the difference between the

Table 5. Number of iterations of algorithm 2 executed on *AUT1* for different values of the Degree of Parallelism *P*

	P				
	1	2	4	6	12
Measured Number of Iterations	69	37	20	14	9
Minimum Number of Iterations	69	35	18	12	6

minimum number of iterations and the measured number of iterations tends to increase. In particular, with 12 testing resources, 9 iterations are needed instead than only 6.

Delay due to the waiting of the termination of the scheduled event sequences. The *runParallelEvents* concurrently starts the execution of at most P test cases on the P available testing resources, and terminates when all the test cases have been terminated and all the descriptions of the obtained GUIs have been returned. The waiting for the termination of all the test cases may represent a waste of time, in particular if the different test cases need different times to be executed. The *Breadth First* scheduling strategy selects for the execution sequences of events having similar length, so the delays due to the different lengths of the sequences of events are small. The Algorithm 2 should be improved by partially refining the *appModel* every time a single GUI description is returned by a *Test Case Executor*, in order to execute the *refineAppModel* operation in concurrence with the *runParallelEvents* operation.

Delay due to concurrency between different Test Case Executor components deployed on the same machine. The deployment of more than one *Test Case Executor* and of more than one AVD on the same machine may cause a reduction of the performance of the tool due to the concurrency between these instances in the access to the resources of the machine. In particular, we have experienced in our experimental configurations that the concurrency of three or more AVDs on the same machines involved in our experiments brings to drastic reductions of the performance of the testing tool that makes these configurations very inefficient. This phenomenon worsen with the most recent Android versions, that needs many more memory resources than the first ones. Technologies such as containers could reduce the weight of this phenomenon.

Efficiency improvements may be obtained by adopting different architectural and technological solutions, too. First of all, the use of real Android devices instead of AVDs may provide an improvement in speed due to the better performance of some devices with respect to the corresponding virtual versions. In particular, by using real devices, it is possible to deploy many *Test Case Executor* components on the same machine having a limited concurrency between them since most of the time spent in the *RunParallelEvents* operation is devoted to the execution of the events on the devices. In this case, the *Test Case Executor* components may also be deployed on the same machine hosting the *Coordinator* component with a consequent saving in terms of number of physical machines.

6 Conclusions and Future Work

This paper proposes a fully automated parallel GUI testing solution for Android applications based on Active Learning techniques. The techniques previously proposed in the literature have been mainly developed having in mind the optimization of their effectiveness, whereas a lesser attention has been paid to their efficiency. The parallel Active Learning algorithm supporting the exploration of the GUIs of Android applications has been obtained by instantiating the Unified Online Testing algorithm proposed in [1] and by modifying it to allow the concurrent execution of some of its operations. The algorithm has been implemented in a prototype tool, whose feasibility and performance have been evaluated against 3 open source Android applications and 6 different deployment configurations. The experimental evaluation has shown that the proposed tool provides a significant reduction of the testing time both with respect to its sequential implementation and with respect to the average reduction of testing time obtained by the PATS tool [28]; moreover, when compared to the ParaAim [7] tool, our tool appears to be more effective. The experiments have shown how the algorithm efficiency depends mainly on the number of deployed machines, whereas an increase in the number of *Test Executor* components per machine does not cause strong improvements in efficiency. This may have happened because the AVDs involved in our experiments shared the same resources on a single machine running concurrently, so we think that this problem can be mitigated by using more machines to host AVDs or exploit real Android devices.

Future work will address the optimization of the algorithm and of the tool and the execution of larger experiments aiming at the evaluation of their scalability. In particular, the migration of the proposed tool from a component based architecture to a service based architecture and its deployment within public cloud infrastructures will be studied and realized.

References

1. Amalfitano, D., Amatucci, N., Memon, A., Tramontana, P., Fasolino, A.: A general framework for comparing automatic testing techniques of android mobile apps. Journal of Systems and Software **125**, 322–343 (2017)
2. Amalfitano, D., Fasolino, A., Tramontana, P., De Carmine, S., Imparato, G.: A toolset for gui testing of android applications. In: IEEE International Conference on Software Maintenance, ICSM. pp. 650–653 (2012)
3. Amalfitano, D., Fasolino, A., Tramontana, P., Ta, B., Memon, A.: Mobiguitar: Automated model-based testing of mobile apps. IEEE Software **32**(5), 53–59 (2015)
4. Amalfitano, D., Fasolino, A.R., Carmine, S.D., Memon, A., Tramontana, P.: Using gui ripping for automated testing of android applications. In: ASE '12: Proceedings of the 27th IEEE international conference on Automated software engineering. IEEE Computer Society, Washington, DC, USA (2012)
5. Anbunathan, R., Basu, A.: Data driven architecture based automated test generation for android mobile. In: 2015 IEEE International Conference on Computational Intelligence and Computing Research, ICCIC 2015 (2016)
6. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. SIGPLAN Not. **48**(10), 641–660 (Oct 2013)
7. Cao, C., Deng, J., Yu, P., Duan, Z., Ma, X.: Paraaim: Testing android applications parallel at activity granularity. In: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). vol. 1, pp. 81–90 (2019)
8. Choi, W., Necula, G., Sen, K.: Guided gui testing of android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. pp. 623–640. ACM (2013)
9. Choudhary, S.R., Gorla, A., Orso, A.: Automated test input generation for android: Are we there yet? (e). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. pp. 429–440 (Nov 2015)
10. Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: Reran: Timing- and touch-sensitive record and replay for android. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 72–81. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013)
11. Griebe, T., Hesenius, M., Gruhn, V.: Towards automated ui-tests for sensor-based mobile applications. Communications in Computer and Information Science **532**, 3–17 (2015)
12. Gu, T., Cao, C., Liu, T., Sun, C., Deng, J., Ma, X., Lü, J.: Aimdroid: Activity-insulated multi-level automated testing for android applications. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 103–114 (2017)
13. Hao, S., Liu, B., Nath, S., Halfond, W.G., Govindan, R.: 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. pp. 204–217. MobiSys '14, ACM, New York, NY, USA (2014)
14. Hu, C., Neamtiu, I.: Automating gui testing for android applications. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 77–83 (2011)
15. Li, A., Qin, Z., Chen, M., Liu, J.: Adautomation: An activity diagram based automated gui testing framework for smartphone applications. In: Proceedings - 8th International Conference on Software Security and Reliability, SERE 2014. pp. 68–77 (2014)

16. Li, X., Jiang, Y., Liu, Y., Xu, C., Ma, X., Lu, J.: User guided automation for testing mobile apps. In: Software Engineering Conference (APSEC), 2014 21st Asia-Pacific. vol. 1, pp. 27–34 (Dec 2014)
17. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 224–234. ESEC/FSE 2013, ACM, New York, NY, USA (2013)
18. Mahmood, R., Mirzaei, N., Malek, S.: Evodroid: Segmented evolutionary testing of android apps. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 599–609. FSE 2014, ACM, New York, NY, USA (2014)
19. Majeed, S., Ryu, M.: Model-based replay testing for event-driven software. In: Proceedings of the ACM Symposium on Applied Computing. vol. 04-08-April-2016, pp. 1527–1533 (2016)
20. Mao, K., Harman, M., Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 94–105. ISSTA 2016, ACM, New York, NY, USA (2016)
21. Meng, Z., Jiang, Y., Xu, C.: Facilitating reusable and scalable automated testing and analysis for android apps. In: Proceedings of the 7th Asia-Pacific Symposium on Internetware. pp. 166–175. Internetware '15, ACM, New York, NY, USA (2015)
22. Nguyen, C., Marchetto, A., Tonella, P.: Combining model-based and combinatorial testing for effective test case generation. In: 2012 International Symposium on Software Testing and Analysis, ISSTA 2012 - Proceedings. pp. 100–110 (2012)
23. Su, T.: Fsmdroid: Guided gui testing of android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 689–691. ICSE '16, ACM, New York, NY, USA (2016)
24. Takala, T., Katara, M., Harty, J.: 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. pp. 377–386. ICST '11, IEEE Computer Society, Washington, DC, USA (2011)
25. Tramontana, P., Amalfitano, D., Amatucci, N., Fasolino, A.R.: Automated functional testing of mobile applications: a systematic mapping study. Software Quality Journal **27**(1), 149–201 (Mar 2019)
26. Tramontana, P., Amalfitano, D., Amatucci, N., Memon, A., Fasolino, A.R.: Developing and evaluating objective termination criteria for random testing. ACM Trans. Softw. Eng. Methodol. **28**(3) (Jul 2019)
27. Wang, P., Liang, B., You, W., Li, J., Shi, W.: Automatic android gui traversal with high coverage. In: Proceedings of the 2014 Fourth International Conference on Communication Systems and Network Technologies. pp. 1161–1166. CSNT '14, IEEE Computer Society, Washington, DC, USA (2014)
28. Wen, H., Lin, C., Hsieh, T., Yang, C.: Pats: A parallel GUI testing framework for android applications. In: 39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2. pp. 210–215 (2015)
29. Zaeem, R.N., Prasad, M.R., Khurshid, S.: Automated generation of oracles for testing user-interaction features of mobile apps. In: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation. pp. 183–192. ICST '14, IEEE Computer Society, Washington, DC, USA (2014)
30. Zein, S., Salleh, N., Grundy, J.: A systematic mapping study of mobile application testing techniques. Journal of Systems and Software **117**, 334 – 356 (2016)