2018 42nd IEEE International Conference on Computer Software & Applications

Exploration Scheduling for Replay Events in GUI Testing on Android Apps

Chia-Hui Lin, Cheng-Zen Yang, Peng Lu, Tzu-Heng Lin, and Zhi-Jun You
Department of Computer Science and Engineering
Yuan Ze University, Chungli, Taiwan, 320
{chl15,czyang,pl16}@syslab.cse.yzu.edu.tw

***Abstract*—Android GUI testing is an important research field to maintain software quality of Android apps. Although many GUI testing schemes have been investigated in the past, the discussion of exploration event scheduling has not been investigated comprehensively for the event replay process. Based on the component priority relationship, this study proposes a novel exploration event scheduling scheme called CPR to reduce the number of the testing events in the replay process. Compared with the breadth-first traversal scheme, the proposed CPR scheme can reduce up to 62% of testing events for achieving the same component coverage, and up to 69% of testing events for layout traversal. Compared with the depth-first traversal scheme, CPR can reduce up to 15% of testing events for achieving the same component coverage, and up to 42% of testing events for layout traversal. With respect to the testing time, CPR can achieve the best performance for most of the AUTs in the empirical study. The results of the empirical experiments show that the proposed CPR scheduling scheme can have the benefits in improving the testing performance.**

I. Introduction

Android is currently one of the most popular mobile operating systems. As shown in the statistics of Statista Inc., the market share of Android based on the Internet use in December 2017 reaches 73.5% [1]. More than 3.7 million Android apps are available in the market in May 2018 [2]. To maintain the software quality, GUI testing plays an important role in discovering software bugs. In the past, many studies have been conducted to automate GUI testing for Android, such as [3]–[12].

In these automatic GUI testing schemes, the GUI ripping technique [13] is commonly used to generate corresponding action events by dynamically analyzing the GUI layouts of an application under test (AUT) [5], [6], [8]–[12]. With GUI ripping, all GUI layouts of the AUT are automatically traversed through the event triggering on all operable GUI components. In the exploration process, the traversal paths construct a tree-like structure called the *GUI tree* from the top-level layout, and the nodes of the GUI tree represent the *UI-states* for the layouts explored. The edge between two UI-states represents a layout transition triggered by issuing an event on a GUI component of the source UI-state. Most ripping systems use a traditional traversal scheme, such as depth-first search (DFS) breadth-first search (BFS), to explore the GUI tree. However, because the UI-states of most AUTs form a directed graph, the ripper needs to check whether the UI-states being explored have been traversed to avoid wasting time in repeatedly visiting the same UI-states. Unfortunately, it cannot be avoided to repeatedly visit many UI-states due to the nature of the traditional DFS/BFS schemes. Therefore, an *exploration event scheduling* problem can be observed in the traversal process for the GUI testing schemes, such as A^2T^2 [4], GUITAR [8], and PATS [10], which use replay operations to explore the GUI tree.

The exploration event scheduling problem is incurred due to two following reasons. First, performing the DFS/BFS traversal needs to backtrack to the parent UI-states having more than one child UI-state. Therefore, one of the following three approaches needs to be used for this backtracking requirement. The first approach is to memorize the snapshots of the parent UI-states in the DFS/BFS traversal process. Unfortunately, taking a snapshot for Android may spend a considerable time, e.g., 8 seconds as reported on Android Studio project site[1]. Moreover, these snapshots cannot be easily migrated among testing devices for parallel GUI testing. Snapshot migration not only lengthens the testing time, but also complicates the system design. The second approach is to use the back operation to return to the parent UI-states. However, the changes in the children UI-states may influence the parent UI-states. Therefore, employing replay operations is the third approach to explore the children UI-states from these parent UI-states. The drawback of the replay operation approach is that it will take much time to reach the destination UI-state from the root UI-state if the replay traversal path is long.

The second reason is that the traditional DFS/BFS schemes do not employ special rules considering the contextual information embedded in the layout structure of a UI-state and the historical traversal information to decide the exploration priorities of GUI components. Therefore, the exploration of DFS/BFS may need to visit many explored UI-states to reach an unseen UI-state. The testing time is thus also lengthened. Since the GUI components of a UI-state in Android apps form a hierarchy (i.e., *UI-hierarchy*), the exploration condition of the UI-hierarchy should be considered in traversal event scheduling to increase the speed of visiting unseen UI-states. If the destination UI-state of a GUI component have been explored, the traversal priority of this GUI component should be decreased to increase the probability of exploring an unseen UI-state in the following replay operations. If a UI-state has many explored GUI components, its traversal priority should be also decreased accordingly.

According to the aforementioned problem, this paper proposes a novel exploration event scheduling approach called CPR (Component-based Priority Ranking) to arrange the ex-

[1] http://tools.android.com/recent/emulatorsnapshots

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00020

ploration events in the replay process based on the properties of GUI components. In CPR, the priority of each exploration event is dynamically calculated according to the contextual information of the GUI components and the explored information of the GUI tree. For each GUI component, a *priority-tree* structure is constructed to derive the exploration priority according to the priority information of its contextual descendants. Therefore, the exploration events of GUI components having lower priorities are deferred based on their priority ranking. Moreover, CPR considers the directed cycles in the GUI tree to reduce the traversal path length. Therefore, the GUI tree is traversed using less exploration events in CPR.

For the event priority problem, Zhu et al. have proposed a context-aware approach called *Cadage* (Context-Aware Dynamic Android GUI Explorer) using a probabilistic event selection scheme to decide the event priority [11]. In Cadage, the event scheduling relies on the frequencies of events performed during the testing. However, Cadage does not deal with the issue of reducing exploration events. In 2017, Yan et al. proposed a GUI exploration scheme called *LAND* (LATTE model generation for Android apps) by considering both component status and the back stack [14]. However, LAND focuses on the issues of code coverage and merging similar UI-states. It does not address the issue of exploration event scheduling in GUI tree traversal. Moreover, both Cadage and LAND do not consider the event replay process in GUI testing.

To evaluate the effectiveness of the proposed CPR scheme, we have conducted empirical experiments with a prototype using five Android apps. Compared with the BFS scheme, CPR reduces up to 62% of testing events for achieving the same component coverage and up to 69% of testing events for layout traversal. Compared with the DFS, CPR reduces up to 15% of testing events for achieving the same component coverage, and up to 42% of testing events for layout traversal. We also measured the average component processing time, the average UI-state processing time, and the average testing time for CPR, BFS, and DFS. The experimental results show that CPR can have the benefits of achieving the best testing performance in most of the AUTs.

The main contributions of this work are as follows:

- We propose a dynamic exploration event scheduling scheme to reduce the number of the testing events in the replay process.
- We present a priority calculation scheme to calculate the event priorities of GUI components based on the priority tree model.
- We have implemented a testing prototype, and performed empirical experiments to demonstrate the improvements.

The rest of this paper is organized as follows. Section 2 provides a brief overview of related work. Section 3 presents the details of the testing system model and the proposed CPR scheduling scheme. Section 4 presents the empirical experiments and discusses the experimental results. Finally, Section 5 concludes this paper and discusses the future work.

II. Related Work

Since the first release of Android in 2008, there have been numerous studies investigating Android automatic GUI testing, e.g., [3]–[12]. This section presents a brief literature review of several representative studies and their techniques about GUI components analysis and event scheduling.

A. Android GUI Testing

In 2011, Hu and Neamtiu purposed an Android GUI test automation environment [3]. They use a Monkey mechanism to randomly trigger events in the AUT. However, the framework does not have a traversal scheme to systematically trigger events for all GUI components. On the contrary, AndroidRipper is a testing automation tool using the GUI ripping technique to generate GUI trees for AUTs [5], [6], [9]. In the ripping process, AndroidRipper uses the traditional DFS/BFS traversal algorithm to explore the GUI tree. Therefore, it achieves a broad testing coverage by traversing all reachable UI-states. However, the traversal approach in AndroidRipper may spend a considerable amount of time in the traversal process [6].

In 2013, Azim et al. proposed two exploration schemes: Targeted Exploration (TE) and Depth-first Exploration (DFE). In TE, dataflow analysis on the app bytecode is performed to construct a control flow graph. Then the graph is systematically explored while the AUT is executed on a real device. On the other hand, DFE uses the depth-first traversal scheme to simulate user behavior. However, they do not discuss the issue of reducing exploration events in the replay process.

For parallel GUI testing, Andlantis is proposed to test many app instances in parallel on a cluster of analysis nodes [15]. Andlantis uses MonkeyRunner to achieve UI exploration. However, the exploration scheme is not explained in detail in [15]. GUITAR is a flexible testing framework that can achieve distributed GUI testing [8]. A DFS scheme is used for ripping in GUITAR to traverse all GUI components. PATS (Parallel Android Testing System) is a parallel testing framework in which the exploration is in a breadth-first manner [10]. Because all UI-states are distributed to the testing slave devices to perform parallel GUI testing, PATS adopts replay operations for exploration. However, both GUITAR and PATS do not address the exploration event scheduling issue.

B. Work of GUI Exploration

For the event priority problem, Cadage is a context-aware approach using a probabilistic event selection scheme to decide the event priority [11]. In Cadage, the event scheduling employs a dynamic GUI model to decide the event sequences by considering the frequencies of the state transitions. It considers two UI-states having the same operable events as the identical UI-states. Our proposed CPR scheme is different with Cadage in two places. First, Cadage calculates the event priorities only based on the frequencies of events performed during the testing. The contextual information of GUI components is not considered. In CPR, the contextual information of each GUI component such as the attributes *text*, *class*, and the children component information, is hash-encoded to denote the contextual status. Second, Cadage only focuses on the traversal condition of each single UI-state. The traversal conditions of

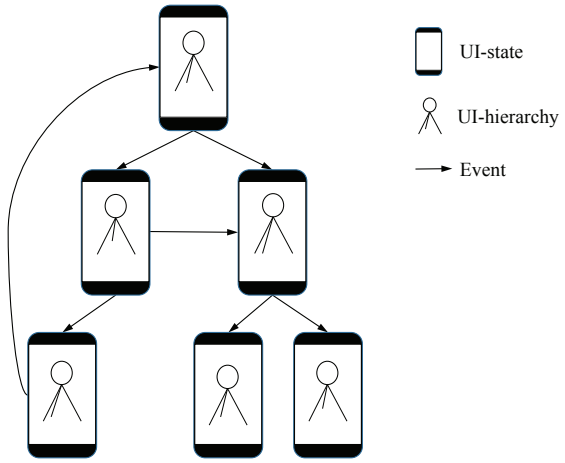


Fig. 1. An example GUI tree of six UI-states having the corresponding UI-hierarchies.

the whole GUI tree are not considered. In CPR, the priorities of the events to the descendant UI-states are considered, and the traversal condition the GUI tree is also investigated.

LAND is a GUI exploration scheme that addresses the identical UI-state problem by considering component status and the back stack [14]. It uses a BFS-based scheme to merge similar UI-states. Therefore, LAND achieves high code coverage based on the back-stack information and the UI-state aggregation model. However, the issue of exploration event scheduling is not considered in LAND. Moreover, instrumentation code needs to be injected into the AUT to get the back-stack information.

III. The Proposed Scheduling Scheme

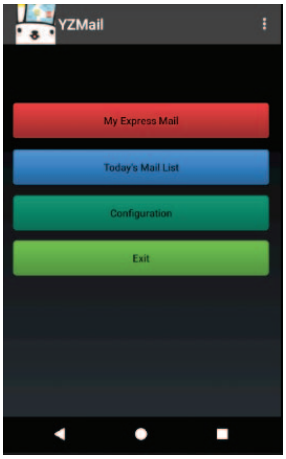
In this section, we first describe the structures in the UI-state model used in this work. The testing system model is thereafter explained to illustrate the testing process. Then the proposed scheduling scheme is elaborated.

A. The UI-State Model

An Android app generally consists of multiple GUI layouts as *Activities*. In this work, each GUI layout is represented as a *UI-state* in which the GUI components form a hierarchy called *UI-hierarchy* to represent the container relationships. The UI-states of an app can be also organized into a hierarchical structure called the *GUI tree*. Each edge in the GUI tree denotes the set of *triggerable events* which is used to perform transitions from one UI-state to another UI-state. Figure 1 illustrates an example GUI tree containing six UI-states with their internal UI-hierarchies.

The XML details of a UI-hierarchy can be obtained from the UI-Automator testing framework. As shown in Fig. 2, the UI-hierarchy XML of the GUI layout in Fig. 2 (a) is extracted as shown in Fig. 2 (b). Based on the extracted UI-hierarchy information, a *Component Table* is used to maintain the status of explored GUI components.

Table I illustrates the structure of the *Component Table* used to record the attributes of GUI components. In the



(a) A GUI layout.

```
<?xml version='1.0' encoding='UTF-8'
   standalone='yes'?>
<hierarchy rotation="0"><node index="0" text=""
   resource-id="" class="android.widget.FrameLayout"
   package="yzu.johnny.yzmail_a"
   content-desc="" checkable="false" ...>
      <node index="0" text=""
         resource-id="android:id/action_bar"
         class="android.view.View"
         package="yzu.johnny.yzmail_a"
         content-desc=""
         ...>
      ...
      </node>
<node index="1" text="My Express Mail"
   resource-id="yzu.johnny.yzmail_a:id/main_my_pack"
   class="android.widget.Button"
   package="yzu.johnny.yzmail_a" content-desc=""
   ... />
...
</node></hierarchy>
```

(b) The XML of the UI-hierarchy.

Fig. 2. An example of the UI-hierarchy of an app.

TABLE I. Component Attributes in the Component Table.

Name	Attribute	Description
ID	string	The hash-encoded string of the attributes in the <node> tag and the contextual children information.
count	integer	The number of visit times of the component.
coordinate	array	The coordinate of the component.
operable	boolean	Used to denote whether the GUI component can be operated.
tested	boolean	Whether the component has been tested.
pri	integer	The event priority of the component.
event	enum	The operable event of the component.

Component Table, the contextual layout information of each GUI component, such as the attributes *text* and *class* in the <node> tag, and the IDs of the children components, is hash-encoded as its ID to represent the component. Therefore, the hash-encoded ID represents the status of the component. The *count* field is used to record the number of the visitations that have happened. If the event of the component has been triggered several times (i.e., $count > 1$), the priority of this event is decreased. The *coordinate* field records the position of the component. If the component is operable, the event

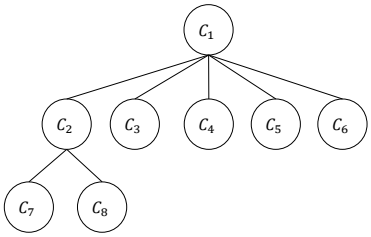


Fig. 3. The priority tree of the app in Fig. 2 (a).

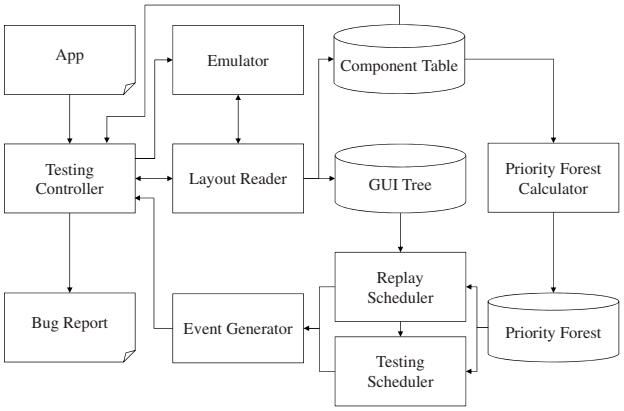


Fig. 4. The system model of the testing architecture.

operation is triggered at this position. The *operable* field is used to denote whether the GUI component has an event to be operated. The *tested* field is used to denote whether the component event has been triggered. The *priority* field records the event priority which is dynamically adjusted.

A *priority tree* can be constructed for each layout of the app according to the component information parsed from the extracted UI-hierarchy. For example, Fig. 3 shows a priority tree of eight GUI components constructed for the layout of the app in Fig. 2 (a). In Fig. 3, C_1 is the FrameLayout containing following seven components. C_2 represents the Action Bar. Since the Action Bar contains an image icon and an overflow button, C_7 denotes the image, and C_8 denotes the overflow button, respectively. C_3-C_6 are the remainder four buttons of the `android.widget.Button` class.

B. The Testing System Model

Figure 4 shows the testing system model in this work. The system model consists of 10 cooperative modules. These modules are elaborated as follows:

- **Testing Controller:** This module controls the overall testing tasks. It is responsible for managing AUTs and test results, and it also controls other modules to perform GUI testing based on the GUI components.
- **Emulator:** The emulator module is responsible for providing an execution environment required for app testing.
- **Layout Reader:** This module is used to extract the UI-hierarchy information. When the system has a newly explored component, the Layout Reader updates the Component Table. It also updates the GUI tree according to the traversal status of the emulator.
- **Component Table:** This is a database to store the information of all explored GUI components. The Testing Scheduler uses the information to determine the event scheduling.
- **GUI Tree:** This is a database to store the explored UI-states. The Replay Scheduler calculates the shortest paths for replay operations based on the information of the GUI Tree.
- **Priority Forest Calculator:** This module is used to calculate the event priorities of the GUI components. When the testing of a GUI component is completed, the priority tree is updated according to the content of Component Table to provide the up-to-date information to the Testing Scheduler.
- **Priority Forest:** This database stores priority forest for future priority calculation.
- **Replay Scheduler:** This scheduler performs replay operations. It determines the shortest traversal path to the destination UI-state according to the priority forest.
- **Testing Scheduler:** This scheduler is used to rank the testing events for future explorations according to their event priorities. The GUI components that have higher priorities are explored earlier.
- **Event Generator:** This module is used to generate testing events to the AUT. It performs replay operations based on the sequence of the events scheduled by the Replay Scheduler. It follows the schedules of the Testing Scheduler to trigger testing events, such as clicks and swipes.

Figure 5 illustrates the processing flow of the cooperative modules in the proposed system model. In the initialization step, the Testing Controller first initializes the testing environment, including AUT installation and extraction of the package name of the AUT. Then, the exploration loop is executed in which the Layout Reader extracts the UI-states of each layout from the emulator to construct its corresponding GUI Tree and update the information to the Component Table. The Priority Forest Calculator thereafter updates the priority forest to adjust the priority values of the nodes.

After the priority forest is updated, the Replay Scheduler decides which UI-state in the GUI tree should be explored next based on the priority weights of the priority forest. The UI-state with a shortest traversal path either from the root UI-state or from the current UI-state becomes the next target UI-state. The Testing Scheduler then selects the component of the highest priority from the selected priority forest of the target UI-state for testing. These two schedulers gather all events into an event sequence to the Event Generator for event triggering. When the event sequence has been triggered, the Testing Controller checks whether all GUI components have been tested.

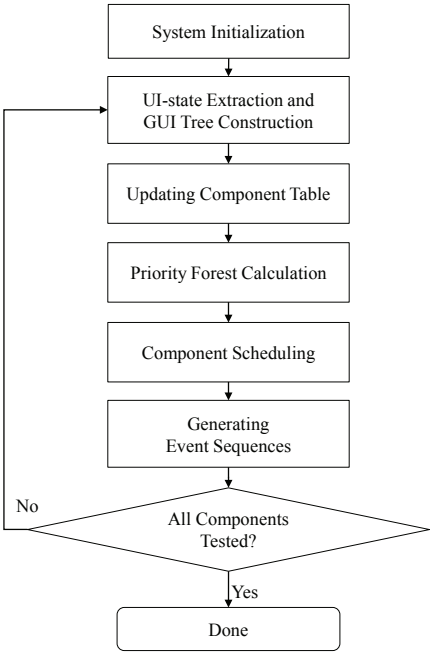


Fig. 5. The processing flow of the testing architecture.

C. The Proposed Scheduling Scheme

The proposed scheduling scheme is performed in two parts: replay scheduling and event scheduling. These scheduling approaches use the GUI tree and the priority forest to schedule the exploration event sequences.

1) Priority Forest Calculation: To perform the scheduling work, CPR constructs the priority forest for explored UI-states and maintains the priority information dynamically. For each UI-hierarchy, a priority tree is constructed to calculate the event priority of each GUI component. In a priority tree, the priority of the parent node is determined from its descendant nodes and its own visitation number. If a GUI component C_i is not operable (i.e., $C_i.operable = False$), its $C_i.pri$ is zero. If C_i is operable, its priority $C_i.pri$ is the sum of all priorities of the children nodes and the local priority $1/(C_i.count)$. Assume C_j is a child of C_i. The event priority $C_i.pri$ of C_i is calculated as follows:

$$C_i.pri = \begin{cases} (\sum_{\forall j} C_j.pri) + \frac{1}{C_i.count} & C_i.operable = T \\ & \wedge\ C_i.tested = F \\ \sum_{\forall j} C_j.pri & \text{otherwise,} \end{cases} \quad (1)$$

The priority of the priority tree ($PT.pri$) is the priority of the root node C_{root}.

$$PT.pri = C_{root}.pri \quad (2)$$

$PT.pri$ is used to decide the set S_{high} in replay scheduling. Since the $C_i.tested$ status of each component is changed dynamically in the testing process, $PT.pri$ is accordingly updated when some $C_i.pri$ is changed in Eq. (1).

Algorithm 1 TestingScheduler: Selecting a component to test

```
1:  function TESTINGSCHEDULER(C_i) // C_i: component
2:      if C_i.operable and !C_i.tested then
3:          C_i.tested = true;
4:          return C_i.event;    // return the event of C_i
5:      else
6:          max_j = 0;
7:          for j = 0; j < C_i.childSize(); j++ do
8:              if C_max_j.pri < C_j.pri then
9:                  max_j = j;
10:             end if
11:         end for
12:         if max_j ≠ 0 then
13:             return TestingScheduler(C_max_j));
14:         else
15:             return null;
16:         end if
17:     end if
18: end function
```

2) Replay Scheduling: Replay scheduling is performed in the Replay Scheduler to reduce the events in the replay process. Because the testing device may have reached a UI-state, both the root UI-state and the current UI-state can be selected as the replay starting point. The replay scheduling scheme then uses Dijkstra's shortest path algorithm to calculate the shortest paths of all unexplored UI-states to the root UI-state and to the current UI-state. The UI-states that have the shortest distance are collected into a set S_{near}. Based on the corresponding priority trees, the unexplored UI-states in S_{near} are ranked according to their $PT.pri$ priorities. The UI-states of the highest priority form a set S_{high} in which the Replay Scheduler randomly selects one as the target UI-state to explore.

When the Replay Scheduler is constructing S_{high}, it also gathers the events for each path to the Event Generator. When the target UI-state is determined, the corresponding event sequence is then triggered. If the maximum $PT.pri$ of the set S_{high} is zero, that means all UI-states are traversed. The Replay Scheduler returns the completion notice to the Testing Controller.

3) Event Scheduling: Event scheduling is performed at the Testing Scheduler to decide the next testing event according to the S_{high} decision and the priority forest. The Testing Scheduler uses `TestingScheduler()` in Algorithm 1 to find an unexplored operable component with the highest priority ($C_{max\_j}.pri$) from the root of the priority tree as the testing target in the next step.

In Algorithm 1, the loop on lines 7-10 is used to check each operable child C_j of C_i to see if it has the highest priority. After the execution of Algorithm 1, the target component is decided, and the Testing Scheduler then obtains the corresponding triggerable testing event for the selected component and sends it to the Event Generator.

IV. Experiments and Results

To investigate the effectiveness of the proposed CPR scheduling scheme, we implemented a prototype and used five Android apps to conduct empirical studies. This section describes the empirical experiments and discusses the results. The

experimental results show that the proposed CPR scheduling scheme can improve the testing efficiency.

A. Research Questions

In the experiments, we studied three research questions for the effectiveness of the proposed CRP scheme.

- **RQ1: Can the proposed CPR scheme quickly traverse the GUI components with less events?** The objective of this research question is to find whether CPR can use less events to test more GUI components. If the number of the exploration events can be reduced, more components can be tested in a limited time period.
- **RQ2: Can the proposed CPR scheme quickly traverse the UI-states with less events?** Since the number of GUI components in each layout may vary greatly, this research question is investigated to find whether the exploration of CPR prefers the layouts having many GUI components. If these dense layouts have higher priorities in the testing process, the number of explored layouts is limited under a short time constraint.
- **RQ3: Can the proposed CPR scheme reduce the replay operation time in the GUI tree traversal?** This research question is studied to show the improvements of reducing the replay operations in CPR. If the number of the replay operations can be effectively reduced in CPR, the testing process of visiting a new layout or exploring a new operable component can be completed as soon.

B. Environment

To conduct the experiments, we collected five apps to study. These five apps include a propitiatory app used to check the express mail status of the post office and four apps of different application domains in open source or in Google Play. To avoid endless exploration in the experiments, each app was required to have a fixed-size GUI tree in the experiments. However, this requirement should not impair the validation of the experiments. These apps in the experiments are as follows:

- **YZUMail** v0.2.a: This app is a propitiatory app used to check the status of the express mail of the local post office. This app has eight UI-states.
- **APV PDF Viewer** v0.4.0[2]: This is an open source PDF viewer supporting textual search and touch gestures.
- **Lightning Expense Tracker** v3.9.2[3]: This app provides a simple expense tracking function. The user can customize the expenditure categories and have convenient billing methods. It has a simple UI design.
- **Yelp** v7.10.1[4]: Yelp provides a global platform with over 100 million store reviews. Users can use it to find nearby shops and restaurants, discover the shops addresses and contact telephone numbers, and give the evaluations.
- **VLC Player for Android** v1.7.2[5]: VLC is a well-known open source multimedia player.

The prototype was implemented using Genymotion 2.12.0 with Google Nexus 4.4.4 API level 19 running on Ubuntu 16.04. The UI Automator framework was used to perform GUI ripping analysis. Due to the inevitable layout transition delays of the emulator, several pause operations were inserted to ensure that the UI information was correctly extracted.

In the experiments, the performance of the proposed CPR scheme was compared with the performance of the traditional BFS and DFS schemes. Because BFS and DFS do not consider the order of the GUI components at the same level in a UI-hierarchy, the components of the same level were randomly selected in the BFS/DFS implementations. The random selection mitigates the influences of the specific ordering of the components in the AUT. Then the performance of all schemes was measured 10 times for each app.

For the performance metrics in the RQ1 study, we measured the number of the events needed to visit a fixed number of unique operable GUI components. This component coverage represents the testing throughput of the GUI components. Therefore, with the same component coverage, a traversal scheme is more time-consuming if it needs to issue more events. The number of the events needed to achieve the same UI-state coverage was measured for RQ2. Similarly, the UI-state coverage was used to demonstrate the testing throughput of traversing unique GUI layouts. This metrics can be used to explore the situation of the replay operations. For RQ3, we measured the average processing time for each unique operable component and for each unique UI-state as the performance metrics. This is mainly because BFS and DFS had different execution sequences in their random selection implementations. Therefore, the total execution time in our BFS/DFS implementation could be variant for each BFS/DFS traversal. The average component processing time t^c_{avg} with respect to all unique operable components is calculated as follows:

$$t^c_{avg} = \frac{t^c_1 + t^c_2 + \ldots + t^c_n}{n}, \tag{3}$$

where t^c_i is the average component processing time of the i-th execution and n is the total number of the experiments. Similarly, The average UI-state processing time t^u_{avg} with respect to all unique UI-states is calculated as follows:

$$t^u_{avg} = \frac{t^u_1 + t^u_2 + \ldots + t^u_n}{n}, \tag{4}$$

where t^u_i is the average UI-state processing time of the i-th execution.

C. Component Coverage

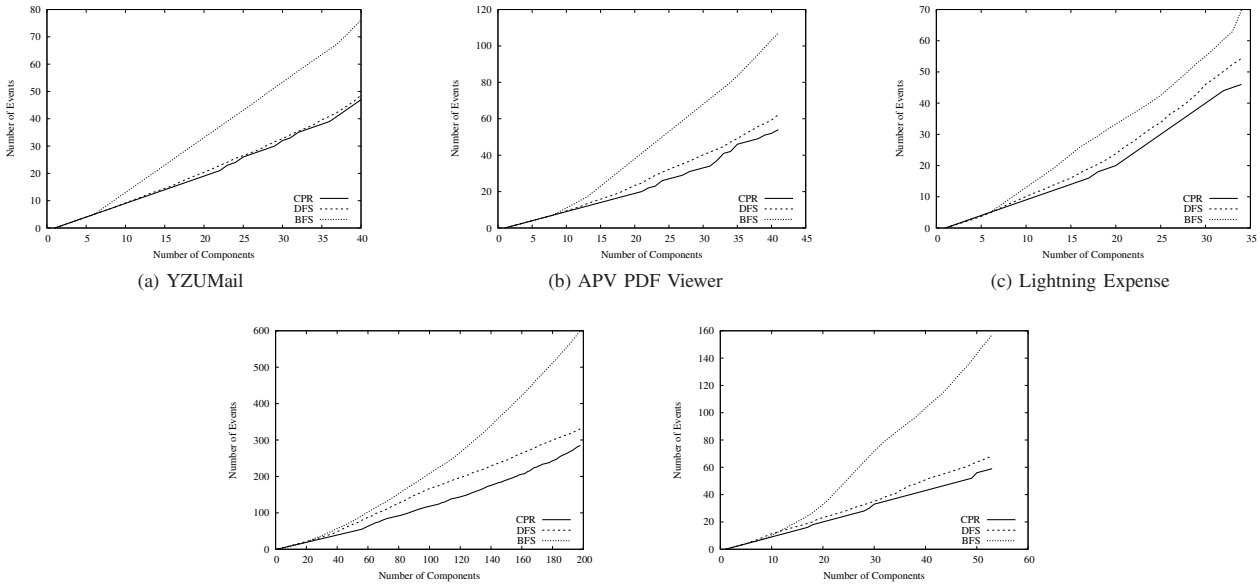
For the research question RQ1, we measured the average number of events to explore the GUI components for the five apps. Figure 6 shows the average number of events under the same component coverage for BFS, DFS, and CPR. The experimental results show that CPR outperforms BFS and

[2]https://github.com/mpietrzak/apv

[3]https://play.google.com/store/apps/details?id=com.maxsuntw.moneytrack

[4]https://play.google.com/store/apps/details?id=com.yelp.android

[5]https://play.google.com/store/apps/details?id=org.videolan.vlc



(a) YZUMail (b) APV PDF Viewer (c) Lightning Expense

(d) Yelp (e) VLC Player

Fig. 6. The average number of events in AUTs under the same component coverage.

DFS. Among three scheduling schemes, BFS consistently has the worst component coverage performance. The reason is that there are many replay operations in the BFS traversal to explore the GUI components in a UI-state. On the contrary, the unexplored GUI components are tested in the future exploration traversal in DFS and CPR. Therefore, DFS and CPR need fewer events to achieve the same component coverage.

Compared with DFS, CPR still needs fewer events to achieve the same component coverage for two reasons. First, DFS needs more replay events when the traversal is at the bottom of the GUI tree. Testing other bottom UI-states incurs more replay events in DFS. Second, CPR always finds the shortest path from two source UI-states: the current UI-state and the root UI-state. If there is a shortest path from the root UI-state to the destination UI-state, CPR uses this path for the future exploration.

Compared with the BFS scheme, the proposed CPR scheme can reduce up to 62% of testing events for achieving the same component coverage. Compared with the DFS scheme, CPR can reduce up to 15% of testing events for achieving the same component coverage.

D. UI-State Coverage

For the research question RQ2, we measured the average number of events to explore the GUI layouts (i.e., UI-states) for the five apps. Figure 7 shows the average number of events under the same UI-state coverage for BFS, DFS, and CPR. This metrics evaluates the performance of layout traversal in each scheduling scheme.

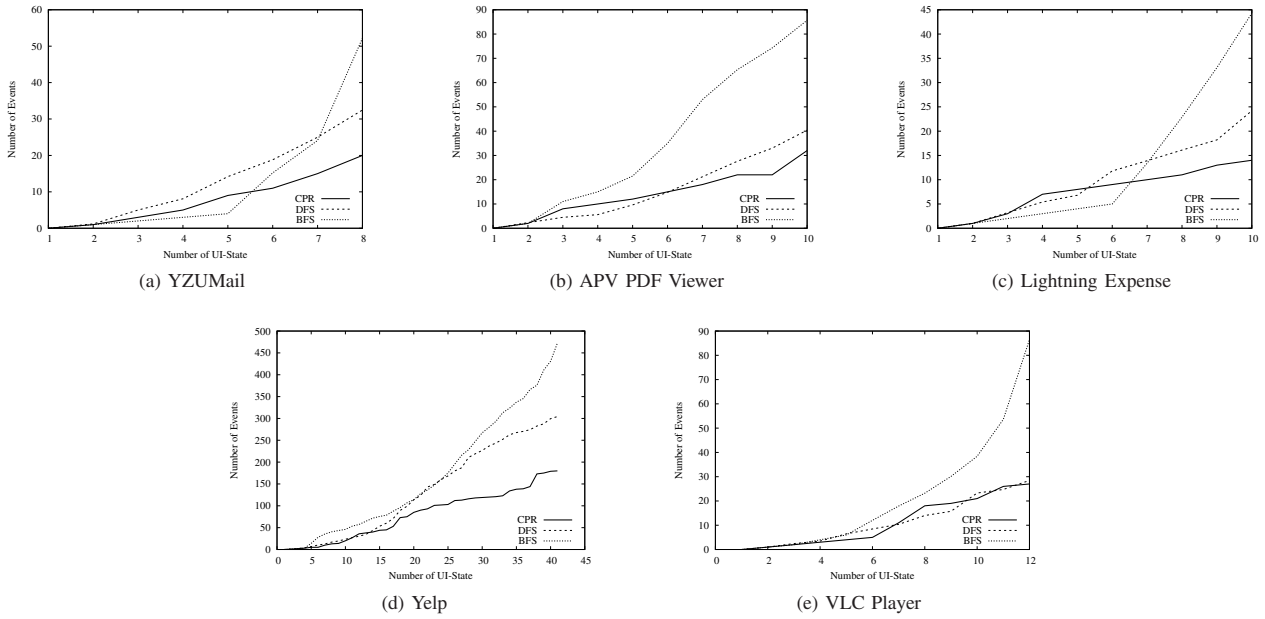
Because each UI-state usually has different number of GUI components, the order of visiting the UI-states highly influences the number of the required events in the scheduling scheme. We can observe that in some apps BFS needs minimum events to discover the UI-states when the number of the UI-states is small. The reason is that the root UI-states of these apps have many children UI-states, and BFS traversal can quickly arrive at these children UI-states with very few replay events in the beginning of the testing process. In the further exploration to the deeper levels of the GUI tree, however, BFS suffers from requiring more replay events. As more UI-states are discovered, BFS performs worse.

In most cases, CPR outperforms DFS because CPR is more likely to explore new UI-states than DFS for two reasons. First, the exploration events of GUI components with lower priorities are deferred in CPR based on their priority ranks. Moreover, CPR uses the contextual exploration information to dynamically adjust the priorities. However, CPR does not perform well in the beginning of the testing process for some apps. The reason is that CPR considers the UI-states in the deeper levels of the GUI tree. Therefore, CPR tends to have more replay events to explore the first few UI-states.

Compared with the BFS scheme, CPR can reduce up to 69% of testing events for achieving the same UI-state coverage. Compared with the DFS scheme, CPR can reduce up to 42% of testing events for achieving the same UI-state coverage.

E. Average Processing Time

To answer RQ3, the experiments were conducted on personal computers running with the same specification of 16GB RAM and an Intel i5-6500 3.7GHz CPU. The results of the average processing time with respect to the number of unique operable components are shown in Fig. 8 in seconds for five AUTs. Because BFS needs more replay operations, its average component processing time is much longer than DFS and CPR. Because DFS has some randomly selected execution sequences which have fewer replay operations in YZUmail, DFS is slightly better than CPR. In other four AUTs, the



(a) YZUMail (b) APV PDF Viewer (c) Lightning Expense

(d) Yelp (e) VLC Player

Fig. 7. The average number of events in AUTs under the same UI-state coverage.

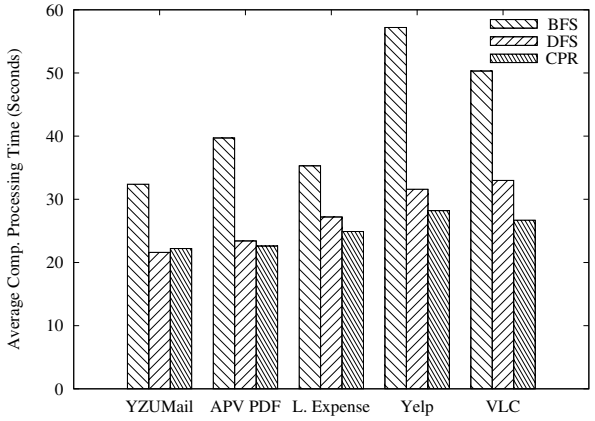


Fig. 8. The average component processing performance in seconds for five AUTs.

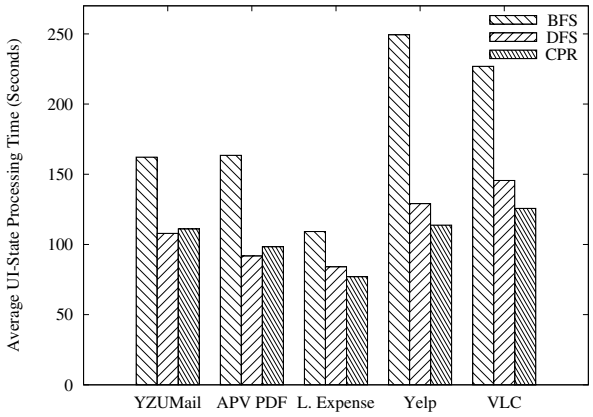


Fig. 9. The average UI-state processing performance in seconds for five AUTs.

experimental results show that CPR has the lowest average component processing time.

Figure 9 shows the average UI-state processing performance in seconds for five AUTs. The experimental results show that BFS has many replay operations, and has therefore the longest average UI-state processing time. Although CPR has a better UI-state coverage performance than DFS for YZUMmail and ADV PDF viewer, DFS achieves the smallest average UI-state processing time in these two AUTs. This is mainly because DFS does not have the UI-hierarchy analysis overhead, and it has the nearly same numbers of replay operations as CPR for these two AUTs. However, in other three AUTS, CPR outperforms BFS and DFS.

Figure 10 shows the average testing performance in minutes for five AUTs. For YZUMail, DFS has the shortest average testing time, and CPR has a comparable performance. For other four AUTS, CPR can finishes the traversal process with the shortest average testing time. Because BFS has many replay operations, its average testing time is the longest for these five AUTs. From the experimental results, we can notice that the benefits of reducing replay operations in CPR are obvious for larger AUTs.

V. Conclusion and Future Work

Android is currently the most widely used smartphone devices. Maintaining the software quality of Android apps is an important research issue. Recently, many GUI testing schemes have been proposed to improve the testing performance. However, the *exploration event scheduling* problem has not been

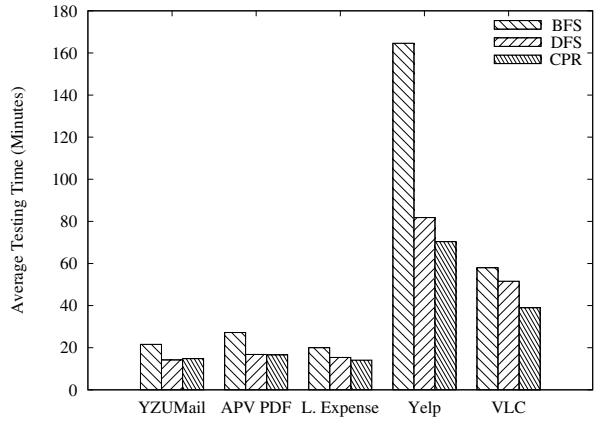


Fig. 10. The average testing performance in minutes for five AUTs.

comprehensively discussed in past related work. Moreover, the contextual information of GUI components are not considered in past studies.

In this paper, we propose a novel event scheduling algorithm called CPR (Component-based Priority Ranking) for replay events in GUI testing. The proposed CPR scheme has two distinct design features. First, a *priority tree* model is proposed to calculate the event priorities based on the contextual layout information and the explored information of the GUI tree. Second, CPR considers the directed cycles in the GUI tree to find the shortest path and thus reduce the traversal path length for replay operations. Therefore, the required replay events can be reduced and new UI-states can be explored as earlier as possible.

To investigate the effectiveness of the proposed CPR scheme, we have implemented a prototype and conducted empirical experiments with five Android apps. Three event scheduling schemes were evaluated: BFS, DFS, and CPR. The experimental results show that CPR can effectively use fewer testing events to achieve the same component coverage and UI-state coverage. For the component coverage metrics, CPR can reduce up to 62% of testing events to BFS and up to 15% of testing events to DFS. For the UI-state coverage metrics, CPR can reduce up to 69% of testing events to BFS and up to 45% of testing events to DFS. With respect to the testing time, CPR can achieve the best performance for most of the AUTs in the empirical experiments. The benefits of reducing replay operations in CPR are obvious for larger AUTs.

Although CPR shows its effectiveness in the current stage, some issues need to be discussed in the future work. First, we have found that some apps have conditional execution paths. For example, when the capacity of the stored data exceeds the predefined threshold, the AUT may enter a UI-state that rarely appears. If these conditional execution paths are dynamically changed at run-time, it is more challenging to maintain the corresponding GUI tree. In these specific cases, CPR could not effectively perform replay scheduling and event scheduling. Moreover, apps may dynamically change the attributes of the GUI components at run time. For example, the background color of a button may be changed according to the different situations. These dynamic changes will complicate the GUI tree processing in identifying the equivalence of the UI-states. More discussions on these practical issues are needed for future improvements.

ACKNOWLEDGMENT

This work was supported in part by Ministry of Science and Technology, Taiwan under grants MOST 104-2221-E-155-004 and 107-2914-I-155-008-A1. The authors would also like to express their sincere thanks to anonymous reviewers for their precious comments.

REFERENCES

[1] Statista Inc., "Mobile operating systems' market share worldwide from January 2012 to December 2017," https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, last accessed on May 20, 2018.

[2] AppBrain, "Number of Android Applications," https://www.appbrain.com/stats/number-of-android-apps, last accessed on May 20, 2018.

[3] C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," in *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*, 2011, pp. 77–83.

[4] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI Crawling-Based Technique for Android Mobile Application Testing," in *Proceedings of the 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011)*, 2011, pp. 252–261.

[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 *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, 2012, pp. 258–261.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and G. Imparato, "A Toolset for GUI Testing of Android Applications," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM '11)*, September 2012, pp. 650–653.

[7] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*, 2013, pp. 641–660.

[8] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An Innovative Tool for Automated Testing of GUI-Driven Software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, 2014.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, September 2015.

[10] H.-L. Wen, C.-H. Lin, T.-H. Hsieh, and C.-Z. Yang, "PATS: A Parallel GUI Testing Framework for Android Applications," in *Proceedings of the 39th IEEE Annual International Computers, Software and Applications Conference (COMPSAC '15)*, vol. 2, July 2015, pp. 210–215.

[11] H. Zhu, X. Ye, X. Zhang, and K. Shen, "A Context-Aware Approach for Dynamic GUI Testing of Android Applications," in *Proceedings of the 39th IEEE Annual International Computers, Software and Applications Conference (COMPSAC '15)*, vol. 2, July 2015, pp. 248–253.

[12] T. Su, "FSMDroid: Guided GUI Testing of Android Apps," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*, 2016, pp. 689–691.

[13] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, 2003, pp. 260–269.

[14] J. Yan, T. Wu, J. Yan, and J. Zhang, "Widget-Sensitive and Back-Stack-Aware GUI Exploration for Testing Android Apps," in *Proceedings of the 2017 IEEE International Conference on Software Quality Reliability and Security (QRS 2017)*, 2017, pp. 42–53.

[15] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe, "Andlantis: Large-scale Android Dynamic Analysis," in *Proceedings of the 3rd Workshop on Mobile Security Technologies (MoST 2014)*, 2014. [Online]. Available: http://mostconf.org/2014/papers/s3p2.pdf