SQDroid: A Semantic-Driven Testing for Android Apps via Q-learning

Hui Guo
School of Computer Science and Technology, Donghua University, Shanghai Key Laboratory of Computer Software Evaluating and Testing
Shanghai, China
guohui@mail.dhu.edu.cn

Xiaoqiang Liu
School of Computer Science and Technology, Donghua University
Shanghai, China
liuxq@dhu.edu.cn

Baiyan Li
School of Computer Science and Technology, Donghua University
Shanghai, China
libaiyan@dhu.edu.cn

Lizhi Cai
Shanghai Key Laboratory of Computer Software Evaluating and Testing
Shanghai, China
clz@ssc.stn.sh.cn

Yun Hu
Shanghai Key Laboratory of Computer Software Evaluating and Testing
Shanghai, China
huy@ssc.stn.sh.cn

Jing Cao
School of Computer Science and Technology, Donghua University
Shanghai, China
caojing@email.dhu.edu.cn

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00041

***Abstract*—Android apps are popular in our daily life, while the testing and maintenance of them is still an open challenge. Random-based testing tools are time-consuming and model-based testing tools are unrealistic to construct all functional behaviors precisely. Existing testing tools based on reinforcement learning not only have difficulty in understanding the business logic of an application, but also face the problem of state explosion during testing. In this paper, we propose SQDroid, a semantic-driven approach for Android apps based Q-learning. SQDroid encourages the Q-learning agent to prefer the actions with functional semantics through a dynamic semantic reward function, which is beneficial to the understand business logic of an app. A state clustering module, employed to compress the state space in Q-table, utilizes the widget attributes on GUI hierarchy to abstract a state. It can reduce the number of states in Q-table and avoid the execution of repetitive actions. We evaluate SQDroid on 48 open-source Android apps. The results show SQDroid outperforms the state-of-the-art model-based/search-based technique/reinforcement learning-based Stoat, Sapienz and Q-testing in terms of code coverage and fault revelation.**

***Index Terms*—Android app testing, Q-learning, dynamic semantic reward, state clustering**

I. Introduction

With the proliferation of smart devices, the number of Android apps has drastically increased [1], [2]. However, how to ensure the quality of applications becomes a great challenge for test engineers. Existing random-based approaches [3], [4] can quickly cover superficial functionalities, but difficult to test complex applications. Search-based approaches [5]–[7] are easily deployed in industrial environments while they generate a large amount of invalid test cases which waste testing time. Although Model-based approaches [8]–[10] achieve a good test result according to modeling the behavior of application, it is almost impossible to design a precise model for each app.

Machine Learning-based approaches also appear in Android automated testing. Recently, several research works [11]–[14] employ reinforcement learning, especially Q-learning [15], [16], to explore apps. They utilize Q-table to record the explored states that abstracted from GUI hierarchy and then update Q-value according to the feedback reward caculated by the differences between two states before and after an action. As the number of exploration increases, the Q-table will describe the actual behavior of an app more precisely. Although existing approaches based on reinforcement learning can be applied to various types of apps without modeling, as trial-and-error approaches, they fall short on generating long and meaningful event sequences for complex apps.

It is observed that one business functionality usually consists of a series of states triggered by the specific event sequence, which we called semantic state sequence. They can inspire the Q-learning agent to test the business logic [17]–[19] of apps. Therefore we propose SQDroid, a novel Q-learning approach based on semantic-driven strategy, which integrates semantic reward and frequency reward to guide the agent during testing, and adjusts the proportion of them according to the consumption situation of semantic state sequences. Under the guidance of this dynamic reward, the agent is preferable to explore functionalities described by the semantic state sequence. As the number of explored semantic state sequences increases, the dynamic reward function adaptively decreases the weight of semantic reward. Supporting by semantic-driven strategy, SQDroid is promising to test hard-to-reach functionalities and reach complex functionalities more early.

State explosion is a very common phenomenon in the reinforcement learning-based approaches [20], [21], which wastes testing time as well as reduces the ability of Q-table to describe app behavior. In order to improve test efficiency,

2693-9177/21/$31.00 ©2021 IEEE
DOI 10.1109/QRS54544.2021.00041

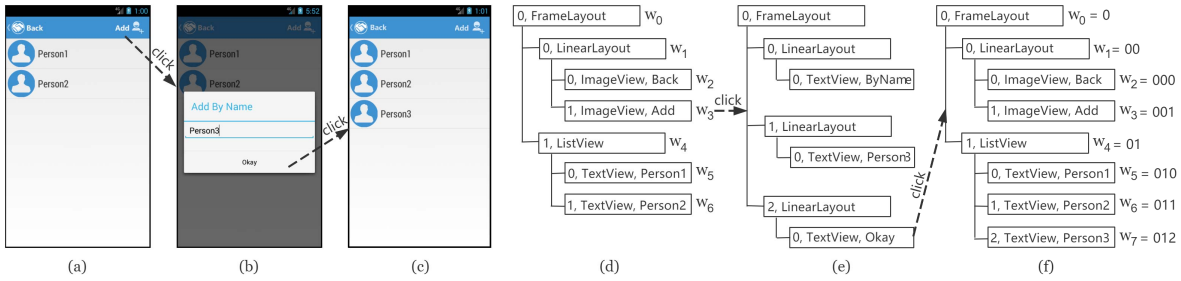

Fig. 1. Fig. 1a, 1b and 1c are three GUI snapshots of Loaned app and Fig. 1d, 1e and 1f are their corresponding GUI trees

we propose a state clustering module to aggregate widgets on the GUI hierarchy and compress the state space in Q-table. As a result, SQDroid avoids repetitively testing some code paragraph and easily tolerates the changes of widgets under diverse layouts (e.g., ListLayout, CircleLayout). Especially for widget-rich apps, it can still precisely build the Q-table against the app behavior.

In this paper, the three main contributions are as follows:

- We novelly propose a semantic-driven exploration strategy based on Q-learning, which employs dynamic semantic reward function to understand the business logic and test complex functionalities in apps.
- We design a state clustering module, which is beneficial to compress state space in Q-table and improve the exploration efficiency of SQDroid.
- We have implemented SQDroid as an Android testing tool and further make a large-scale evaluation. Results show that SQDroid outperforms existing approaches in terms of both code coverage and crash detection. We also make the tool and data available on Github: https://github.com/androidAppGuard/SQDroid.

The remainder of this paper is structured as following: Section II introduces necessary background about Android App and Q-learning; Section III makes an introduction to our semantic-driven exploration based Q-learning; Section IV describes the state clustering module; Section V evaluates our approach and shows the experiment results; Section VI summarizes previous related works about automatic testing technologies and Section VII makes a conclusion.

II. Background

This section provides relevant information about reinforcement learning-based Android GUI testing.

A. GUI of Android App

An Android app interacts with users by showing various Graph User interface (GUI) pages. There are many widgets on the GUI page and they are arranged into a tree-like structure, which is called GUI tree in this paper [8], [20]. Usually, a widget is a text box (e.g., TextView, EditText), button (e.g., RadioButton, ImageButton) or container layout (e.g., FrameLayout, LinearLayout) and it can trigger GUI actions (e.g., click, long-click and swipe) to execute the corresponding code. A widget is described by a series of attributes (e.g., text, index and class) and each attribute is a key-value pair. Figure 1 shows some GUI snapshots and their corresponding GUI trees before and after the operation of "Add" one person. We use w_i to denote a widget located in the i-th node from top to bottom in the GUI tree. The text "Person2" in Fig. 1a is widget w_6 in Fig. 1d and its text attribute is Person2. All the widgets $w_0, w_1, w_2, w_3, w_4, w_5, w_6$ in Fig. 1d form the GUI page in Fig. 1a.

B. Q-learning

Q-learning [22] is control-temporal difference Learning and model-free Policy prediction, inspired by behaviorist psychology, which aims to learn perform optimally for maximizing the cumulative reward in an unexperienced environment. The agent utilizes a trial-and-error way to interact with environment and receives a delayed reward signal after each interaction. By repeatedly experiencing each action in every state, the best action overall will be performed when facing existing situations.

Mathematically, the reinforcement learning problem can be ideally formulated as a Markov decision process (MDP) which is officially defined as 4-tuple $< S, A, P, R >$. Where S is the set of all non-terminal states and A refers to the set of all actions. Figure 2 shows the "agent-environment" interactions in Markov decision process. For each interaction, the agent observes own state $s_t \in S$ in the environment then selects and performs an action $a_t \in A$ according to adopting the policy π^*. At the next time step t+1, the agent receives a numerical immediate reward $r_t \in R(s_t, a_t)$ and enters a new state s_{t+1}. This process will happen repeatedly until exceeding the limit time.

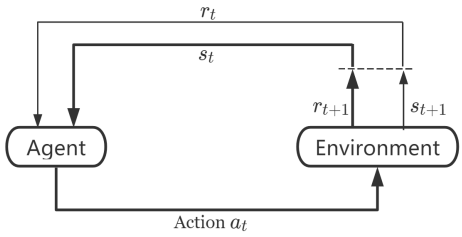

Fig. 2. The "agent-environment" interactions in MDP

In Q-learning, the estimation of how good a state-action pair is defined by Q-value function which returns the expected

cumulative reward of any combination of states and actions after performing a sequence of actions. In order to further describe the value of state-action pair in adjacent interactions, we use equation 1 to iteratively estimate the value of the Q-function.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (1)$$

In this equation, α is learning rate ($0 \leq \alpha \leq 1$) which influences the speed of state-action pair value propagation from current observation to subsequent observation. It is worth to emphasize that γ can balance the relation between immediate and cumulative reward. A bigger γ means more cumulative reward while smaller γ means more immediate reward. To maximize the expected cumulative reward, the optimal policy π^* can represent the action with the highest value Q^* in every state. If the agents explores all actions in all states sufficiently, the equation 1 will make the estimator converge to the true Q-value according to a strict proof [22].

The Q-learning algorithm can learn an optimal action selection policy in Markovian environment and the interaction can be viewed as a MDP which artificially sets reward signals. Base on this, we employ Q-learning to explore Android apps for maximizing the code coverage and discovering more crashes.

III. Semantic-Driven Exploration Based Q-learning

A. MDP for Android Testing

In this paper, SQDroid mathematically models Android testing problem as an MDP, which can be defined with 4-tuple $< S, A, P, R >$. The definition of S, A, P and R for the Android testing is as follows.

S: States. The goal of defining a state is to distinguish different GUI pages on an app correctly. Given a GUI page, SQDroid extracts all widgets on its corresponding GUI tree to combine a state. The state $s_t = (w_0, w_1, \cdots, w_n)$ is an n-tuple where w_n is the n-th widget contained in the GUI tree. For example, Fig. 1d is the corresponding GUI tree of Fig. 1a, which is composed of seven widgets. Therefore the state of page Fig. 1a can be represented by $(w_0, w_1, w_2, w_3, w_4, w_5, w_6)$. In order to identify each widget and refrain state explosion, SQDroid utilizes the bound attribute b and index attribute i of a widget, ignoring text attribute, to form a 2-tuple (b, i). Then the state in Fig. 1a can be described as $((b_0, i_0), (b_1, i_1), (b_2, i_2), (b_3, i_3), (b_4, i_4), (b_5, i_5), (b_6, i_6))$. In brief, our state s_t is defined as a combined state $((b_0, i_0), (b_1, i_1), \cdots, (b_n, i_n))$ where (b_n, i_n) is the tuple of the index attribute and the bound attribute corresponding to the widget. Finally, we use the hash value to substitute the n-tuple, which aims at accelerating the matching efficiency when searching for states in Q-table.

A: Actions. In this paper we define the events of app as actions of MDP and there is no difference between events and actions. Similar to previous research [11], [12], SQDroid analyzes the widget attribute (e.g., click, longClick, scroll) to obtain the available events and defines them as actions in current state. One difference is that we utilize bound attribute b and action type t to combine an action tuple (b, t) which can be directly converted into executable adb commands for faster interaction.

To balance the relationship between "exploit" and "explore", SQDroid adopts $\epsilon - greedy$ policy to select an action in the current state s_t (described in equation 2). In the case of probability $1 - \epsilon$, the agent chooses the action with the highest value for exploiting the performed actions and selects a random UI action with probability $\frac{1}{2}\epsilon$ for exploring new state. In order to found intricate crashes, the agent also chooses a system-level action (e.g., volume-up, power, camera) to simulate the realistic scenarios. Generally, SQDroid sets 0.2 as the default value of ϵ.

$$selectAction(s) = \begin{cases} \max_a Q(s, a) & 1 - \epsilon \\ random\ a\ system\ action & \frac{1}{2}\epsilon \\ random\ a\ UI\ action & \frac{1}{2}\epsilon \end{cases} \quad (2)$$

P: Transition Function. The transition function is used to describe which state the app will jump to after an action is executed, and it is determined by the application.

R: Reward. The reward is critical for Android testing, because it has decisive impacts on the exploration strategy. Existing technologies do not consider the semantic information of the exploring state sequence and they are hard to explore complex functionalities. So SQDroid designs a dynamic reward function, which both considers the business logic of exploring state sequence and the execution frequency of the action.

B. Semantic State Sequences

As described above, one state can be abstracted from a GUI page of app, then a state sequence can be used to represent the GUI page of sequential experience after performing a series of actions. To identify whether a state sequence implements a functionality, SQDroid proposes the semantic state sequence, which should fulfill the following three conditions:

- Every state in the state sequence belongs to the state space of Q-table.
- Only one action can be performed (ignoring text editing) when transitioning from the previous state to the next state.
- A state sequence implements at least one business functionality.

Figure 3 shows an example of semantic state sequence, which describes the partial states and actions of an app called Book Catalogue. The app is used to store and manager a list of books. We list three semantic state sequences to explain corresponding business functionality. As the diagram shows, there are six states $s_1, s_2, s_3, s_4, s_5, s_6$ in the app and two of the states may be transformed by one action such as the

state s_1 jumps to s_2 after performing the action a_2. In this diagram, $[s_1, s_2, s_1]$, $[s_1, s_3, s_5, s_3]$, $[s_1, s_3, s_6, s_3]$ are three semantic state sequences since each of them achieves one business functionality by executing a series of orderly actions. $[s_1, s_2, s_1]$ adds a new book and the business functionality of "adding a book" is executed. $[s_1, s_3, s_5, s_3]$ and $[s_1, s_3, s_6, s_3]$ view the book details at the beginning while the difference between them is that $[s_1, s_3, s_6, s_3]$ edits the book information instead of deleting the book item.

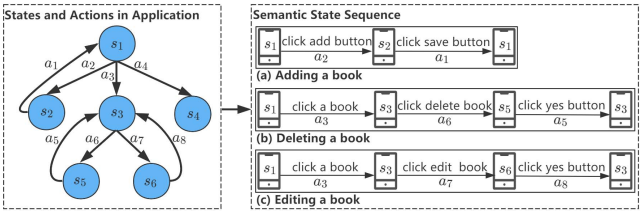

Fig. 3. Semantic state sequence in Book Catalogue app

Semantic state sequence is similar to the use case but not exactly the same. They all implements at least one business functionality and can be easily obtained from unit testing, integration testing, crowdsourced testing or provided by developers. However, the semantic state sequence records the app states after action is executed while the use case saves the actions that require to be performed. Besides, the semantic state sequence ignores text editing because SQDroid is concerned with states rather than actions.

The semantic state sequence supports the calculation of semantic reward, so SQDroid can realize the semantic-driven exploration and understand the business functionality of apps.

C. Approach Overview

SQDroid contains two stages, semantic state sequences collecting and Q-learning exploring. The task of semantic state sequences collecting is to collect semantic data and Q-learning exploring utilizes them to guide the agent to explore apps. The overall workflow of SQDroid is depicted in Figure 4 and we make a detailed description as follows.

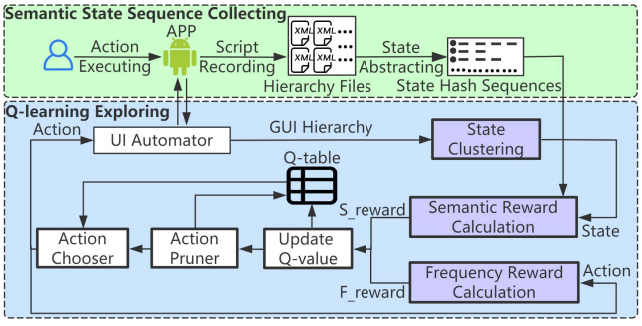

Fig. 4. SQDroid workflow

Semantic state sequences collecting. We first collect the basic use cases according to official function description of app, and then these use cases are executed automatically on the app under testing according to a script. Besides executing each action in use cases, the automatic script also saves the response GUI hierarchy files of the app. According to the definition of state, SQDroid abstracts the corresponding state from every GUI hierarchy file, and these states obtained by each use case can orderly form a semantic state sequence. After gaining all the semantic state sequences, SQDroid uses hash value to substitute each state in the semantic state sequences, which can reduce the storage space of state and the time consumed for searching. Finally, these hash sequences will be loaded into memory for further semantic reward calculation.

Q-learning exploring. SQDroid utilizes the loaded state hash sequence to iteratively guide the agent toward semantically-rich functionalities. The process can be viewed as MDP where the App is the environment. In the beginning of each exploration, SQDroid dumps the GUI Hierarchy of app's page through UIAutomator then employs the state clustering module to obtain the current state. In addition, the state will be orderly stored in an exploring state sequence, which is used for reward calculation. After updating the state sequence, the corresponding semantic reward (S_reward) and frequency reward (F_reward) can be work out from the state hash sequences in the stage of semantic state sequences collecting and the number of execution times of the action.

As the number of explored semantic state sequences increases, SQDroid will dynamically adjust the proportion of each part of the final reward. The reward is used for updating the Q-value of the prior state-action pair (s_t, a_t) in the Q-table according to equation 2. When an unexplored state is observed, all the related state-action pairs are added to Q-table and each state-action pair is assigned an initial value. To avoid unnecessary exploration, Action Pruner will set the value of the action to 0 if there is no change between the previous state s_t and the current state s_{t+1}. Finally, Action Chooser chooses an action with maximum Q-value or randomly selects a system-level action from Q-table in the current state. The app will give a response to the action a_t and start the next exploration.

D. Dynamic Reward Function

SQDroid proposes a dynamic reward function, besides frequency reward, it imports semantic reward based on semantic state sequences to explore business functionalities. Furthermore, the weight of each reward can be dynamically adjusted to balance the "exploit" and "explore" during Q-learning exploring.

Semantic reward. SQDroid defines the semantic reward r_s as equation 3. P represents the set of collected semantic state sequences and Q is an exploring state sequence in one episode. The set of continuous subsequences of Q is defined as $sub(Q)$. If a sequence seq belongs to both P and $sub(Q)$, it indicates that the current testing has explored the semantic state sequence seq in P. The number of such sequence is described as $|\{seq \,|\, seq \in P \wedge seq \in sub(Q)\}|$. To scale the reward value to between 0 and 1, it will be divided by the total number of semantic state sequences $|P|$.

$$r_s = \frac{|\{seq \,|\, seq \in P \wedge seq \in sub(Q)\}|}{|P|} \tag{3}$$

We give an example to illustrate the specific calculation process. Suppose Q is $[A, B, C]$ and P is [[AB],[AC],[BC]] where each letter is a state of the app under testing, then the continuous subsequences $sub(Q)$ is $[[A], [B], [C], [AB], [BC], [ABC]]$ by enumerating Q. For $[AB]$, $[BC]$ belong to both P and $sub(Q)$, therefore the number of explored semantic sequences $|\{seq \,|\, seq \in P \wedge seq \in sub(Q)\}|$ is 2 and the semantic reward is $\frac{2}{3}$. This formula considers the relationship between the continuous subsequences of the exploring state sequence and the semantic state sequences. The more semantic state sequences are explored, the more meaningful functionalities are tested.

Frequency reward. SQDroid uses r_f to represent frequency reward which is defined as equation 4. The function $f(s, a)$ calculates the execution frequency of action a in the state s, while r_f is inversely proportional to the value of $f(s, a)$. We utilize a global buffer to save the number of execution of an action for calculating r_f. The initial value of r_f is 1, an action is performed more times, it will become smaller. With the encouragement of frequency reward, the agent can systematically experience all available actions.

$$r_f = \frac{1}{f(s, a)} \tag{4}$$

Dynamical combination reward. To fully unleash the potentials of the two rewards, SQDroid designs a dynamical weight to combine them. As shown in equation 6, the function $g(P)$ is used to obtain the total number of explored semantic state sequences so far, and a global variable is used for storing the information about whether a semantic state sequence has been explored. SQDroid regards $\frac{g(P)}{|P|}$ as the weight of r_f while the remaining ratio $1 - \frac{g(P)}{|P|}$ is viewed as the weight of r_s. Our key insight is that the agent can adaptively adjust the exploration strategy according to the consumption situation of collected semantic state sequences.

$$r = (1 - \frac{g(P)}{|P|}) * r_s + \frac{g(P)}{|P|} * r_f \tag{5}$$

We use an example to illustrate the impact of the dynamical combination reward on exploration strategy during the testing. At the beginning, the agent has never visited a semantic state sequence so that the weight of r_s is the maximum value of 1 while the weight of r_f is 0. As a result, the explore strategy will prioritize actions with the semantic reward. As the agent visits more semantic state sequences, the weight of r_s becomes smaller and it does mean that the agent will tend to explore unexecuted actions.

E. Semantic-Driven Exploration

Different from previous exploration strategies [6], [9], [11], we propose a semantic-driven exploration which first encourages the testing tool to gradually understand existing business logic and then explores more complex functionalities. It can efficiently cover code with business functionalities and reveal logical crashes.

Algorithm 1: Semantic-Driven Exploration

input : application under test AUT, maximum use case length l, exploration time t, initial Q-value Q_{init}, learning rate α, semantic state sequences $S_{semantic}$
output: test suite T

1 $Q \leftarrow \Phi$ ▷ Initialize Q-table
2 $T \leftarrow \Phi$ ▷ Initialize test suite
3 $t \leftarrow \Phi$ ▷ Initialize test case
4 $S_{explored} \leftarrow \Phi$ ▷ Initialize explored state sequence
5 **while** $\neg$*isTimeover(t)* **do**
6 $s_t, actions_t \leftarrow analyzingGuiInfo()$
7 **foreach** $action_t \in actions_t$ **do**
8 **if** $action_t \notin Q$ **then**
9 $setQ\_value(Q,\ s_t,\ action_t,\ Q_{init})$
10 **end**
11 **end**
12 $a_t \leftarrow selectAction(Q)$
13 $s_{t+1} \leftarrow executeAction(AUT, a_t)$
14 $r_s \leftarrow getSemanticReward(\{r_{explored} \cup s_{t+1}\}, S_{semantic})$
15 $r_f \leftarrow getFrequencyReward(a_t)$
16 $r \leftarrow getCombinationReward(r_s, r_f)$
17 $Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma Q^*(s_{t+1}, a) - Q(s_t, a_t))$
18 $S_{explored} \leftarrow S_{explored} \cup s_t$
19 $t \leftarrow t \cup a_t$
20 $pruneAction(Q, s_t, a_t, s_{t+1})$
21 **if** $getLength(t) \geq l$ **then**
22 $T \leftarrow T \cup t$
23 $(S_{explored}, t) \leftarrow (\Phi, \Phi)$
24 $restartAUT()$
25 **end**
26 **end**

Algorithm 1 describes the pseudocode of semantic-driven exploration. It requires six input parameters: 1) application under test, 2) maximum length of test case, 3) exploration time, 4) initial Q-value for new actions, 5) learning rate and 6) semantic state sequences. With the help of these parameters, SQDroid executes a series of actions and return a test suite as an output. Specifically, we maintain a memory array to save previously explored state (line 4, 18) for each new test case. This array helps SQDroid to identify which semantic state sequence has been executed and calculate the semantic reward. Before generating each test case, SQDroid firstly analyzes the GUI hierarchy to abstract the current state and executable actions through state clustering module (line 6). For each action in the state, the Q-table will record its value or assign an initial value if it has never been performed (line 7-11). Then SQDroid employs equation 2 to infer the next action to be

executed (line 12). After performing the action, the agent will reach a new state, the corresponding semantic reward (line 14) and frequency reward (line 15) can be calculated by equation 3 and equation 4. SQDroid utilizes equation 5 to dynamically combines the two rewards (line 16). To improve calculation efficiency, every state in the exploring state sequence and the loaded semantic state sequences will be replaced by the corresponding hash value.

It is worth emphasizing that the exploration benefits from the propagation of Q-value (line 17), which enables the agent to choose valuable actions no matter what the current action is. After an action is executed, if the new state gains higher reward value r_t, the Q-value $Q(s_t, a_t)$ of the corresponding action will also increase and it is more possible to be performed in next exploration. Otherwise, the new state with lower reward value will decrease the priority of action. In order to strengthen the effect of new states, we set γ to 0.99 for more efficient exploration. During the testing, the value of action which leads to crashes or terminations will be set to 0 to avoid the futile exploration (line 20).

In order to replay the explored scenarios, all executed actions forms a test case (line 19) and the current test case will be merged into the global test suite if the length of current test case is greater than user-specified value (line 20-24). Before the next exploration, the explored state sequence and the test case will reinitialize to an empty set.

In brief, under the influence of the propagation property of Q-value and the incentives of dynamical combination reward, the agent will try to experience collected semantic state sequences as much as possible in the first stage due to the larger weight of the semantic reward, and it will guide the agent to understand business logic. The next stage it derives high-quality test input to cover unexplored functionalities by increasing the weight of frequency reward.

IV. State Clustering Module

A. State space in Q-table

According to the definition of state in section III, SQDroid utilizes the widgets contained in the GUI hierarchy tree to define a state. Take Figure 1 as an example, the state in Fig. 1d is $s_0 = (w_0, w_1, w_2, w_3, w_4, w_5, w_6)$ and the state in Fig. 1f is $s_1 = (w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7)$. s_1 has one more TextView widget w_7 (Person3) than s_0 and they are two different states in Q-table. However, their two corresponding snapshots (Fig. 1a and Fig. 1c) both show the list of personnel information and clicking on these TextView widgets all jump to the details page of the person. In terms of programme code, the clicking actions of the TextView widgets execute the same code paragraph. If taking s_1 as a new state, some small changes (e.g., the relative position of widgets and the number of widgets) will generate a lot of new states. It will produce a huge state space in Q-table.

The huge state space is disruptive to the update of Q-value and reduces the testing efficiency. It needs a lot of time for the agent to perform all the actions in each state, let alone precisely construct a Q-table that describes all the behaviors of the app. As a result, the actions selected based on Q-table may not increase code coverage and trigger more crashes. In order to tackle this problem, we propose a state clustering algorithm.

Algorithm 2: State Clustering Algorithm

```
1  Function getState(GuiHierarchy)
2      root ← getDomtree(hierarchy)
3      {curIndexPath, pathList} ←
         {String(), List()}
4      recursiveAnalysis(root, curIndexPath, pathList)
5      S ← Set()
6      foreach indexPath ∈ pathList do
7          if indexPath[: −1] ∉ S then
8              S.insert(indexPath[: −1])
9          end
10     end
11     return S.hashcode()
12 end
13 Function recursiveAnals(node, curIndexPath,
     pathList)
14     curIndexPath ←
         curIndexPath + node.indexAttr
15     pathList.append(curIndexPath)
16     saveAvailableActions(node)
17     foreach cNode ∈ node.childNodes do
18         recursiveAnals(cNode, curIndexPath, pathList)
19         curIndexPath ← curIndexPath[: −1]
20     end
21 end
```

B. State Clustering Algorithm

The type-same widgets (e.g., w_5, w_6, w_7 in Fig. 1f) in a same layout container will mislead SQDroid to define many new states in Q-table and these widgets have the same hierarchical relationship. Analogy to hierarchical clustering method, SQDroid regards the GUI tree as a hierarchical cluster tree where the widgets contained in each layer are a cluster. In order to compress state space, SQDroid utilizes the index attribute of each widget and its relative position on the GUI tree to define the widget, and then the parent widget of each cluster is used to represent this cluster. Finally, all the clusters are combined as a state.

Taking Figure 1 as an example, SQDroid obtains the index attribute path of each widget (such as the paths of w_5, w_6 and w_7 in Fig. 1f are 010, 011 and 012) by $recursiveAnals$ function and the detailed paths are marked on right side in the GUI tree. As we can see, these widgets w_5, w_6, w_7 are in same hierarchical level and they are regarded as a cluster. We observed that w_4 is the parent widget of the cluster, and the index attribute path 01 of w_4 is same as the index attribute path of w_5, w_6 and w_7 when the last index of their paths is removed. Therefore SQDroid employs the index attribute path of w_4 to define the cluster for aggregating these wid-

gets. For other clusters, SQDroid adopts the same processing steps and the state is composed of all clusters. For example, there are four clusters in Fig. 1f and their original index attribute paths are $[0], [00, 01], [000, 001], [010, 011, 012]$. The w_0 is root widget, so this cluster has no parent widget and the remaining three clusters form the corresponding state. According to the $getState$ function, three parent widgets of these clusters are defined as the state $[0, 00, 01]$. The detailed procedure is described in algorithm 2.

This algorithm takes the GUI hierarchy as input and returns hash code of the state. The GUI hierarchy is parsed as a rooted, ordered dom tree where each node is a widget with a set of attributes such as index attribute and bounds attribute (line 2). According to the tree, SQDroid deploys a depth-first traversal to analyze each node (line 4), and then the relative index attribute path of each node is sequentially appended to the list $pathList$ (line 14-20). Meantime, nodes whose event attributes (e.g. click, longClick, scroll) are true will be saved as available actions in current state (line 16). After the recursion is complete, it inserts each element (the index attribute path of each node) of the path list to the set collection S (line 6-10) and returns the hash code of the S as the state hash (line 11).

We abandon the last digit of each index attribute path to form the S (line 7), which means that the index attribute paths of these widgets in same level are equable. Aggregating these widgets as a cluster is beneficial to reduce the number of state and improves the accuracy of Q-table.

V. Evaluation

In this section, we mainly evaluate SQDroid in two aspects. One is the ability to increase code coverage and reveal more crashes; the other is the ability to enhance the testing efficiency. We seek to answer the following research questions:

RQ1: Code Coverage. How is the code coverage achieved by SQDroid when comparing with the state-of-the-art and the state-of-practice testing tools?

RQ2: State Clustering Module. Is SQDroid able to improve the ability to compress state space and testing efficiency when employing this state clustering module?

RQ3: Fault Revelation. How is the fault revelation ability of SQDroid when comparing with the state-of-the-art and the state-of-practice testing tools?

A. Implementation

Tool Implementation. SQDroid is implemented as a automated app testing framework, which reuses and extends the following tools: the GUI hierarchy files of applications are dumped by Android UI Automator; the executable actions are analyzed by Python library DOM; the action execution commands (including system-level actions) are delivered by Android Debug Bridge (ADB). So far, SQDroid supports UI actions (click, long click, edit), navigation actions (scroll, back, menu, home) and system-level actions (open browser, add volume, caps lock, screencap). For code coverage collection, SQDroid monitors open-source applications by Jacoco and we also enable Stoat and Sapienz to compute code coverage with Jacoco for fairness.

Execution Environment. We conduct all experiments on a physical machine of Ubuntu 16.04, which is composed of hardware facilities 4 cores 3.20GHz CPU and 8GB RAM. All experiments are conducted on Android Kitkat version with API 19 (Android 4.4), the same as publicly available Sapienz and Stoat, and each emulator is configured with 2GB RAM.

Target Apps. We collect 48 open-source Android Applications as our benchmark. The mainly source is related work [11], [15]. However, some applications cannot be compiled due to no maintenance for a long time. We select 23 apps and most of them have less than 1K executable lines of codes (ELOC). To make our experiments more convincing, we add 25 larger apps with complex logical functionalities to enrich our benchmark. These apps are from other source apps list [23] and F-Droid [24]. For diversity category applications may reflect the advantages of semantic-driven exploration strategy and the number of target apps has a balance between several common categories such as news, finance, tools and communication.

Semantic State Sequence Data Collection. According to the definition of the semantic state sequence, we pick out the basic use cases according to official function description of target apps because the semantic state sequence and the use case both achieve at least one functionality. In order to transform these use cases into semantic state sequences, we utilize a script to perform all the actions in use cases and record the corresponding GUI hierarchy file for each action. The collected data is used for calculating semantic reward. In Table I, The column "#Data" shows the number of collected semantic state sequence for each app. We collected a total of 222 semantic state sequences for 48 target apps and the length of each semantic state sequence is in the range of 3-8.

B. Evaluation Setup

To answer the research questions above, we conduct three empirical studies on 48 target apps.

Study 1. To answer RQ1, we compare SQDroid with Monkey [3], Stoat [8], Sapienz [5] and Q-testing [11], which are regarded as the state-of-practice and state-of-art tools in Android automated testing, to investigate how much code coverage is improved. According to their previous research work, we limit the deadline for each experiment to 1 hour. For Monkey, we assign 200 milliseconds to wait the response of the app under the testing. For Stoat, we set 30 minutes for modeling and 30 minutes for generating test input. For Sapienz, we adopt default population size 50 to evaluate each app. Several apps cannot be executed due to the large population size, so we adjust the population size of these apps to one with the highest code coverage. As for Q-testing, we adopt the default configuration due to it is a closed-source testing tool.

We also compare $SQDroid^1$ (employing dynamical combination reward as reward function) with $SQDroid^2$ (employing frequency reward as reward function) to investigate whether

TABLE I
TESTING RESULTS ON 48 OPEN-SOURCE APPS

Subject		%Coverage								#Crashes							#States		#Data
Name	ELOC	M	SA	ST	Q	S	SQ^1	SQ^2	SQ^3	M	SA	ST	Q	SQ^1	SQ^2	SQ^3	SQ^1	SQ^3	S
MunchLife	232	81	82	81	43	27	80	76	77	0	0	0	0	0	0	0	8	10	3
Mynotes	267	84	90	82	56	59	92	75	77	0	1	0	1	1	0	0	7	12	4
RecordTimeDroid	301	70	89	88	32	12	89	89	80	0	0	0	0	0	0	0	9	12	3
Snotepad	347	74	84	81	50	21	96	80	79	0	0	0	0	0	0	0	18	21	5
Manpages	388	3	5	4	2	2	5	4	3	0	1	0	0	1	1	1	10	10	4
AnyCut	402	34	69	31	22	40	68	64	67	2	3	2	1	5	0	2	5	8	3
Manille	502	13	16	15	15	15	15	14	14	1	0	0	0	1	0	0	8	6	5
BatteryDog	513	72	75	66	76	19	69	68	67	0	0	0	0	0	0	0	4	4	4
DumpPhone	570	59	41	41	31	32	41	41	32	0	1	0	0	1	1	0	4	5	3
SoundBoard	629	33	33	48	33	30	33	30	33	1	1	3	2	1	1	3	9	11	4
LockPattern	741	8	11	8	8	7	9	8	8	2	2	2	2	2	0	2	9	12	5
Siggen	751	96	95	87	86	73	82	78	82	2	0	0	1	1	0	0	5	9	5
MultiSmsSender	852	10	8	7	7	3	8	7	7	0	0	0	0	0	0	0	9	10	4
Whohasmystuff	853	80	82	79	13	58	79	13	63	0	1	0	0	0	0	0	15	19	5
AutoNight	856	28	30	31	20	24	28	28	25	2	2	3	2	3	2	1	3	6	3
ALogcat	889	74	76	53	75	57	74	73	72	0	0	0	0	0	0	0	8	8	5
SmsScheduler	929	6	8	9	6	6	10	6	9	2	2	0	0	0	0	0	20	22	4
Alarm	996	19	20	20	19	15	23	19	20	1	3	1	1	3	1	1	15	9	5
Talarmo	1020	75	73	74	74	63	74	77	74	2	0	1	2	1	0	1	7	7	4
ToneDef	1433	22	41	40	38	31	39	39	39	1	0	2	1	2	2	0	9	11	5
PasswordMaker	1511	31	19	21	13	14	17	14	14	2	1	1	0	1	0	0	2	6	4
MoneyBalance	1725	32	38	13	32	30	69	51	39	0	1	0	0	1	1	0	27	36	5
Notes	1864	57	58	52	39	20	47	38	41	5	2	0	0	0	0	0	25	31	4
ImportContacts	1987	8	7	7	7	5	8	7	7	1	0	1	1	6	0	1	9	11	3
Loaned	1988	57	30	65	33	39	59	64	46	0	0	0	1	1	1	1	20	43	7
BudgetWatch	2390	45	48	49	39	29	53	46	45	1	1	3	1	2	1	1	52	34	6
Budget	2470	65	58	58	54	50	71	66	67	0	1	0	0	1	0	0	61	46	5
GoodWeather	2499	52	50	65	51	21	71	49	67	2	4	0	0	5	2	1	34	28	6
XiaExpress	2897	40	12	12	12	11	12	12	11	0	0	0	0	0	0	0	4	4	4
BeQuick	2922	48	50	38	21	38	21	33	37	1	0	1	1	3	0	0	9	15	4
Swiftp	3106	28	31	31	31	27	31	31	31	2	2	2	2	0	2	0	15	15	5
Ammeter	4096	25	28	27	28	25	27	22	25	0	2	1	4	5	0	0	8	10	5
Sanity	4964	15	13	7	5	7	12	7	12	1	0	1	0	1	0	1	10	17	5
Tomdroid	5044	39	39	40	43	20	54	40	44	1	1	0	1	2	1	2	27	25	5
RadioBeacon	7483	45	39	41	39	28	42	39	40	4	1	3	1	3	3	3	28	108	5
Materialistic	8464	40	61	40	32	47	63	62	40	4	2	0	1	6	5	5	90	38	5
KeepassDroid	9369	18	16	17	9	8	17	11	11	0	0	0	0	0	0	0	8	7	4
Vanilla	10717	43	35	43	47	20	50	47	50	5	2	0	3	5	5	1	67	88	5
BetterBatteryStats	10862	30	21	32	33	12	14	12	12	0	0	0	0	0	0	0	7	9	5
APhotoManager	11691	39	35	40	31	9	25	10	12	3	4	1	1	7	0	1	75	12	4
Timber	11876	30	30	24	22	13	38	27	30	4	2	0	0	4	4	2	111	151	5
AnyMemo	12119	33	55	16	57	15	59	27	55	0	0	1	0	2	0	1	125	145	5
RunnerUp	15119	40	21	38	21	22	52	47	48	1	2	0	1	2	2	1	75	74	6
AmazeFileManage	24223	28	28	28	30	19	30	28	29	2	2	4	4	6	6	2	210	296	6
BookCatalogue	24858	28	29	26	12	7	34	26	30	3	3	1	4	5	4	3	38	47	5
Suntimes	30197	28	42	18	13	19	43	35	33	3	2	3	1	9	2	3	101	141	6
Anki	35556	29	32	22	38	19	38	29	33	1	2	1	2	7	0	1	76	91	5
MyExpenses	41171	32	33	22	31	25	42	17	39	1	4	3	1	11	0	3	192	272	5
Average/Sum	6388	40	41	38	32	24	44	37	39	63	58	41	43	117	47	44	35	42	222

the semantic reward can improve the code coverage and the ability to reveal crashes. The dynamical combination reward in section III considers both the semantic relationship of the exploring state sequence and the execution frequency of actions while the frequency reward only involves the number of actions performed. We implemented the dynamical combination reward into $SQDroid^1$ and the frequency reward into $SQDroid^2$. For each testing tool, we start from 0 to calculate the achieved code coverage. To alleviate randomness, we run five repeated experiments on the 48 target apps and investigate how much average code coverage has been achieved.

Study 2. To answer RQ2, we compare $SQDroid^1$ (implemented by state clustering module) with $SQDroid^3$ (implemented by combining executable widgets) to investigate the code coverage and the number of states involved. The previous state abstraction algorithms [15], [16] in reinforcement learning usually utilize executable actions to define a state, therefore we implemented this algorithm into $SQDroid^3$.

Study 3. To answer RQ3, we compare SQDroid with Monkey, Stoat, Sapienz and Q-testing to investigate how many crashes are found during the testing. In order to obtain the number of crashes, we analyze the output log of logcat and aggregate these crashes for each testing tool to get unique crashes.

C. Experimental Results

RQ1: Code Coverage. Table I shows the average instruction coverage of state-of-the-art testing tools (M, SA, ST, Q, S, SQ^1 indicate Monkey, Sapienz, Stoat, Q-testing, semantic state sequences and $SQDroid^1$) on 48 target apps. We sort the apps by ELOC extracted from Jacoco report and the gray cells indicate which tool achieves the highest average instruction coverage. As we can see, SQDroid outperforms the state-of-practice Monkey (40.54%), the state-of-the-art search-based technique Sapienz (41.37%), the model-based technique Stoat (38.27%) and the reinforcement learning-based Q-testing (32.06%) by covering 44.37% average instruction coverage. The average instruction coverage (24.5%) obtained by performing the semantic state sequences is much less than SQDroid. It also shows that SQDroid has the ability to explore uncollected functionalities. At the same time, We found that SQDroid achieves the highest instruction coverage for 23 apps while Monkey, Sapienz, Stoat and Q-testing are 9, 13, 6 and 5. In addition, we group these apps by the size of ELOC and calculate the final coverage results which is depicted in Figure 6. It can be seen from these box-plots, SQDroid achieves the highest average instruction coverage in four app size groups.

The column SQ^1 (denotes $SQDroid^1$) and the column SQ^2 (denotes $SQDroid^2$) in Table I also show the test result of SQDroid whether to use the semantic reward as the reward function. For the average instruction coverage, SQ^1 with the semantic reward and the frequency reward is 44.37% while SQ^2 with the frequency reward only is 37.20%. For the number of crashes, SQ^1 reveals 117 and also achieves higher performance than SQ^2 (47). The result means that the semantic reward improves the ability of SQDroid to cover more code and reveal more fault.

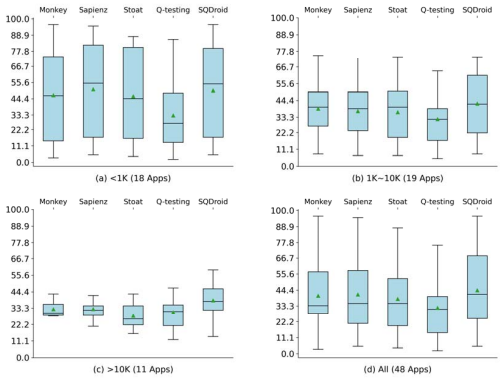

Fig. 5. Instuction coverage achieved by Monkey, Stoat, Sapienz, Q-testing and SQDroid

RQ2: State Clustering Module. In Table I, SQ^1 represents $SQDroid^1$ (implemented by state clustering module) and

SQ^3 denotes $SQDroid^3$ (implemented by combining executable widgets). In term of code coverage and fault revelation, SQ^1 achieves better performance than SQ^3. As for code coverage, SQ^1 achieves 44.37% on average while SQ^3 is 39.08%. For fault revelation, SQ^1 discovers 117 crashes in total while SQ^3 is 44. We also record the number of all states in the Q-table during the testing for each app, as we see column "#State" in Table I, SQ^1 define less states than SQ^3 and therefore avoids a lot of repetitive explorations.

In addition, we manually compare the number of states and activities in several apps, and find that a few activities correspond to a lot of states where most of them are similar and unnecessary. For example, we obtain 36 activities from the source code of app MyExpenses while there are 272 states defined in the Q-table of SQ^3. It is a fatal shortcoming for testing tools based on reinforcement learning, because massively similar states require the agent to re-execute all the actions in each state. However, according to the comparison of the number of SQ^1 and SQ^3 states, we note that SQ^1 effectively reduces the number of state in Q-table.

RQ3: Fault Revelation. Table I also shows the unique crashes for each testing tools. We adopt the definition in Stoat, a fault is defined as a crash or exception (containing the keywords of the app's package name) in stack traces. SQDroid, Monkey and Sapienz reveal the most crashes on 29, 12 and 10 apps, respectively. In the remaining apps, 7 apps have the most crashes detected by Stoat while the other 4 apps have the most crashes detected by Q-testing. In total, SQDroid detects 117 crashes among all the apps, which outperforms Stoat (41), Q-testing (43), Sapienz (58) and Monkey (63).

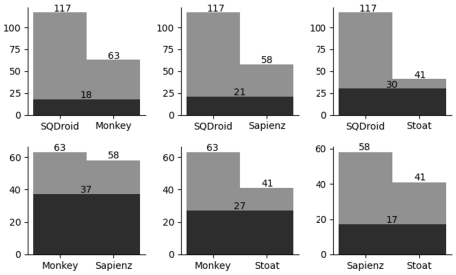

Fig. 6. Pairwise comparison on revealing crashes

To investigate whether the testing tools are complementary on revealing crashes, we make a pairwise comparison whose result is shown in Figure 7 (the proportion of crashes revealed by both testing tools is the dark grey areas). SQDroid reveals the largest number of total crashes on 48 apps while the highest proportion of the number of overlapping crashes is found by Monkey and Sapienz. We think the reason for this phenomenon is that Monkey and Sapienz both randomly inject events to generate event sequences. As for unique crashes, SQDroid discovers 77 while Monkye, Sapienz and Stoat find 23, 15 and 8. It means the ability of SQDroid to reveal unique crashes can not be replaceed by other testing tools.

D. Threats to Validity

Internal Threats. The main threat to internal validity is the choice of parameter value. Values of different input parameters have a significant impact on testing results. To mitigate the issue, we follow the default value for Monkey, Sapienz and Stoat as much as possible. For several apps that do not apply the default values, we choose the best performance parameter configuration according to conducting small-scale experiments.

Additionally, some crashes are caused by the low performance of the Android emulator. The device may become unresponsive due to a large number of events sending in a short period of time. To avoid the problem, SQDroid and monkey set an interval of 200ms in two events and we also rule out the crashes caused by the low performance of devices.

External Threats. The main threat to external validity is semantic state sequence data collection. Since we collect the semantic state sequences of core business logic by artificial decision, some critical semantic state sequences may be ignored and cause the incorrect reward. To alleviate the threat, we decide to choose the basic use cases through the official description of app. If the use case can achieve at least one app functionality, we adopt it and translate it into the semantic state sequence; otherwise we abandon it.

VI. Related Work

Many technologies have been proposed for Android automated testing. We categorize and briefly introduce existing automatic testing techniques according to different exploration algorithms.

Random-based Technologies. One of most practical testing tool randomly selects an available user event or system-level event to fuzz the apps under testing. Monkey [3], which is integrated in the android system, sends massive pseudo-random event streams to simulate user operations. Although the high efficiency and the ability to quickly detect the robustness and stability of apps, it falls short on testing complex applications due to the relatively unintelligent event generation manner. From the point of view, SQDroid considers event execution frequency as frequency reward to systematically select event. Thus, many futile testing can be avoided.

Search-based Technologies. The first search-based testing tool EvoDroid [7] utilizes experienced event and state to construct a graph and employs the graph to guide the search process. Different to it, Sapienz [5] leverages the first Pareto multi-objective (e.g., code coverage, test sequence length) to test apps, and significantly outperforms Monkey and Dynodroid on 68 open-source real-world apps. They have aroused a lot of industry attention while the probability of generating available test cases is unpromising. SQDroid selects the executable events based on the current state, so each event is available to the app.

Model-based Technologies. Model-based testing technologies [9], [25] are popular in Android automated testing. Stoat [8] builds a dynamic weighted UI model during the exploration and the test cases are generated by Gibbs sampling on the

model. A3E [26] employs the targeted exploration (directly and quickly exploring the target activities) and the depth-first exploration (systemically exploring the activities) to test apps. To some extent, SQDroid benefits from model-based technologies because the update of Q-table is based on the actual behavior of app and then the Q-table can be viewed as the model of app.

Learning-based Technologies. The most closely related technologies leverage reinforcement learning to explore apps. Adamo et al. [15] and Vuong et al. [16] first employ Q-learning to guide the testing process. They design frequency reward function and difference reward funtion to influence the agent's exploration strategy. Unfortunately, only simply relying on this information, the strategy cannot achieve high code coverage in complex apps. However, none of them considered the business logic of the application and thus their performance is limited. SQDroid tackles this problem by designing a dynamic semantic reward function to understand business logic of complex functionalities.

VII. Conclusion and Future work

In this paper we propose SQDroid to explore more complex functionalities for Android app. Additionally, SQDroid aggregates the type-similar widgets to avoid state explosion. The state space reduction improves the accuracy of Q-table describing app behaviors. To verify the validity of SQDroid, we conduct a thorough evaluation on 48 open-source target apps and the evaluation results show that SQDroid outperforms the state-of-the-art search-based technique Sapienz and the best model-based technique Stoat. In the future we will evaluate SQDroid on some real bugs data set [27].

Acknowledgment

We thank the anonymous QRS reviewers for their valuable feedback. We also thank Ting Su from East China Normal University for his constructive comments on the early draft of this work. This work was supported by Shanghai Key Laboratory of Computer Software Evaluating and Testing.

References

[1] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 408–419.

[2] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes understanding and benchmarking framework-specific exceptions of android apps," *IEEE Transactions on Software Engineering*, 2020.

[3] Google, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, 2021.

[4] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[5] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[6] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 481–492.

[7] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[8] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[9] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[10] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 469–480.

[11] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[12] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 105–115.

[13] Y. Koroglu and A. Sen, "Reinforcement learning-driven test generation for android gui applications using formal specifications," *arXiv preprint arXiv:1911.05403*, 2019.

[14] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *2011 IEEE fourth international conference on software testing, verification and validation workshops*. IEEE, 2011, pp. 252–261.

[15] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8.

[16] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31–37.

[17] E. Scornavacca, S. J. Barnes, and S. L. Huff, "Mobile business research published in 2000-2004: emergence, current status, and future opportunities," *Communications of the association for information systems*, vol. 17, no. 1, p. 28, 2006.

[18] R. M. Müller, B. Kijl, and J. K. Martens, "A comparison of inter-organizational business models of mobile app stores: There is more than open vs. closed," *Journal of theoretical and applied electronic commerce research*, vol. 6, no. 2, pp. 63–76, 2011.

[19] C. S. Leem, H. S. Suh, and D. S. Kim, "A classification of mobile business models and its applications," *Industrial management & data systems*, 2004.

[20] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 238–249.

[21] 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, 2014.

[22] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[23] pcqpcq, "pcqpcq/open-source-android-apps: Open-source android apps," https://github.com/pcqpcq/open-source-android-apps, 2021.

[24] F.-D. Limited, "F-droid - free and open source android app repository," https://f-droid.org/zh_Hans/, 2021.

[25] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[26] 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 & applications*, 2013, pp. 641–660.

[27] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 119–130.