A Reinforcement Learning-based Approach to Testing GUI of Moblie Applications

Chuanqi Tao
taochuanqi@nuaa.edu.cn

Nanjing University of Aeronautics and Astronautics

Yuemeng Gao
Nanjing University of Aeronautics and Astronautics

Hongjing Guo
Nanjing University of Aeronautics and Astronautics

Jerry Gao
San Jose State University

Research Article

Keywords: Reinforcement Learning, GUI testing, Advantage Actor Critic, Proximal Policy Optimization, Finite-State Machine

Posted Date: February 13th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-2565153/v1

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at World Wide Web on February 22nd, 2024. See the published version at https://doi.org/10.1007/s11280-024-01252-9.

A Reinforcement Learning-based Approach to Testing GUI of Moblie Applications

Chuanqi Tao[1,2,3,4*], Yuemeng Gao[1], Hongjing Guo[1] and Jerry Gao[5]

[1*]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing,China.
[2]Ministry Key Laboratory for Safety-Critical Software Development and Verification, Nanjing University of Aeronautics and Astronautics, Nanjing,China.
[3]Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing,China.
[4]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing,China.
[5]Computer Engineering Department,San Jose State University, USA.

*Corresponding author(s). E-mail(s): taochuanqi@nuaa.edu.cn;
Contributing authors: gaoyuemeng@nuaa.edu.cn;
guohongjing@nuaa.edu.cn; Jerry.gao@mail.sjsu.edu;

Abstract

With the popularity of mobile devices, the software market of mobile applications has been booming in recent years. Android applications occupy a vast market share. However, the applications inevitably contain defects. Defects may affect the user experience and even cause severe economic losses. This paper proposes ATAC and ATPPO, which apply reinforcement learning to Android GUI testing to mitigate the state explosion problem. The article designs a new reward function and a new state representation. It also constructs two GUI testing models(ATAC and ATPPO) based on A2C and PPO algorithms to save memory space and accelerate training speed. Empirical studies on twenty open-source applications from GitHub demonstrate that: (1) ATAC performs best in 16 of 20 apps in code coverage and defects more exceptions; (2) ATPPO can get higher code coverage in 15 of

20 apps and defects more exceptions; (3)Compared with state-of-art tools Monkey and ARES, ATAC, and ATPPO shows higher code coverage and detects more errors. ATAC and ATPPO not only can cover more code coverage but also can effectively detect more exceptions. This paper also introduces Finite-State Machine into the reinforcement learning framework to avoid falling into the local optimal state, which provides high-level guidance for further improving the test efficiency.

Keywords: Reinforcement Learning, GUI testing, Advantage Actor Critic, Proximal Policy Optimization, Finite-State Machine

1 Introduction

With the rapid development of mobile computing and wireless network technology and the widespread use of mobile devices in the past few years, the scale of mobile applications also has been overgrown. To meet the demands of users and conform to the trend of The Times, mobile applications (APPs) need to be developed and iterated constantly. Many applications may contain defects. Due to the different versions of devices and operating systems, mobile apps often need help with cross-platform and cross-version compatibility. If the APP fails, it may lead to poor user experience and even huge financial losses. For example, the China Merchants Securities and Huaxi Securities trading systems went down in 2022, causing economic losses for investors. In the same year, Chengdu's nucleic acid testing system collapsed, causing a waste of medical resources and citizens' time resources. Therefore, how to test these applications to ensure the correct operation of the APP has become the key to solving the compatibility and security at this stage.

There are many operating systems for mobile devices, such as Android, iOS, Harmony, Blackberry, Symbian, etc. In terms of mobile devices, Android devices occupy a massive share of the mobile market, and the number of Android apps is also increasing. Therefore, this paper studies the Graphical User Interface (GUI) test of Android apps. Android apps are mainly written in Java and stored in executable files in the form of dex files. The APP is distributed as an apk file. This example contains the dex file, code (if any), and other resources. An APP declares its main components in the AndroidManifest.xml. There are four main elements: activities, services, broadcast, receivers, and content providers. An activity is a component responsible for the GUI. An activity corresponds to a UI interface, which includes many UI elements (such as menus, buttons, and images). Developers can control the behavior of activities by implementing callbacks for each lifecycle (for example, create, pause, and destroy). The activity responds whenever the user responds to an interface action (such as a click), which is the primary goal of the Android testing tools. The service component can run for a long time in the background. Unlike activities, there is no user interface, so they are not usually the direct target of Android testing tools, although they may be tested indirectly through some activities.

The GUI may contain many widgets, and the GUI testing mainly performs functional testing on the application under test (AUT). GUI testing checks the behavior of the APP by interacting with the GUI (for example, clicking, long clicking, scrolling, and typing strings). If the behavior of an APP deviates from what is expected, the APP contains some defect. However, with the continuous development and iteration of APP, the composition of APP becomes more and more complex, and checking its function and behavior may require some clarification. Due to limited human resources and time pressures, Android APP GUI testing is expensive. New challenges arise when trying to replace human testing with machine tools. These challenges include the explosive growth of state combinations and the limitations of exploration. Automated testing of Android APPs has great research potential.

GUI testing has attracted massive attention from researchers. Some researchers propose random testing, which generates random events on the GUI. One of the most famous random test tools is Monkey [1], which Google provides for stability and stress testing. It generates random user events such as key presses and random inputs. However, random tests like Monkey may generate large amounts of invalid and redundant events. It is ineffective for them to explore more states and detect failures. Model-based strategies [2–6] build precise or abstract GUI models by static or dynamic methods to generate test cases. Nevertheless, the strategies are influenced by two problems. One of the problems is inherent state explosion. Another one is that the effectiveness of generated test cases depends on the built model's integrity and the representation of the application's state.

Reinforcement learning (RL) contains agent learning strategies to maximize returns or achieve specific goals by interacting with the environment. It is extensively used in Android GUI testing. Unlike supervised learning, which requires labeled datasets, RL can learn automatically by interacting with the environment. Though RL is applied to GUI testing, most are implemented using Q-learning. Q-learning uses a table to record expected values of actions that are called action values in a specific state. The Q table occupies a lot of memory. Some researchers are considering replacing tabular methods with Deep Neural Networks (DNN). The agent utilizes DNN to learn the action-value function automatically through past experiences. Driven by a reward function, the agent guides us to explore the AUT, unlike random testing, which explores the AUT without purpose. The optimal action to perform can be predicted by the agent even if the state has never been visited before. It can effectively solve the state explosion problem. RL allows us to test GUI effectively and efficiently. DeepGUIT [7] adopts Deep Q Network (DQN) to represent the value function, and ARES [8] applies TD3, DDPG, and SAC to fit value functions. However, these methods utilize a replay buffer that records the pairs of states and rewards. The replay buffer also occupies a lot of memory, even if less than Q-table. In reinforcement learning, exploration is one of the most challenging issues. In a mobile application, a function is triggered by a specific operation sequence. As the length of the operation sequence becomes more extended, exploration becomes more challenging.

In this paper, we propose **ATAC**(**A**utomatic **T**esting based on **A2C**) and **ATPPO**(**A**utomatic **T**esting based on **PPO**). They are novel approaches based on deep reinforcement learning to test Android GUI. ATAC applies the A2C algorithm to Android GUI testing to avoid using a relay buffer. A2C algorithm that introduces the idea of parallelism constructs multiple threads, and the thread interacts with the environment, respectively. Besides, ATAC does not need a replay buffer, and it can save memory space and consume fewer resources. The neural network it constructs is much smaller than DeepGUIT and ARES. It can work even if the agent does not equip with GPU. The PPO algorithm is an improvement of the A2C algorithm. It uses the importance sampling method to convert the off-policy into the on-policy strategy, which improves data utilization. ATPPO applies the PPO algorithm to Android GUI testing to avoid using a relay buffer. To reduce interruption of the execution sequence, we consider constructing a Finite-State Machine(FSM) during the test process. If the strategy falls into the local optimal state, according to FSM, it provides more advanced guidance for exploring reinforcement learning algorithms. We evaluate them in twenty apps in the environment of Android 10. Experimental results show that our approaches achieve higher coverage and detect more failures than the state-of-the-art test generation tool Monkey and RL-based ARES.

To summarize, this paper has the following major contributions:

- We utilize executable GUI elements in XML files to represent states.
- We adopt a new reward function to avoid sparse rewards, and the reward function in this paper mainly depends on the change of state and whether errors can be detected.
- We construct two GUI testing models(ATAC and ATPPO), respectively, based on the A2C and PPO algorithms.
- We put forward the exploration strategy based on FSM, which provides high-level guidance for further improving test efficiency.
- Empirical studies on 20 applications have found that our approaches have significantly different effects on the code coverage and the number of defects detected.

The remainder of this paper is structured as follows: Section 2 introduces relevant basic concepts and definitions. In Section 3, our proposed approach is described in detail. Section 4 shows the evaluation of our approach and discusses the results. Then, the threats to validity are shown in Section 5. Section 6 gives a general description and summary of related work. Finally, section 7 concludes our work.

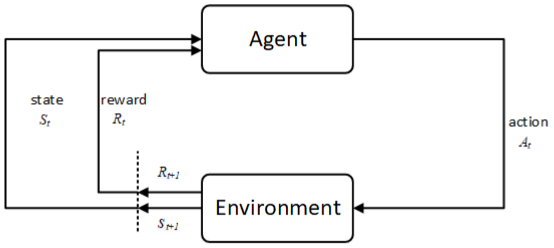

Fig. 1 Markov decision process

2 Preliminaries

Section 2.1 introduces the concept of reinforcement learning and the specific Markov decision process. There are two main methods to solve the Markov decision process: model-based and model-free methods. The model-based approach is mainly implemented by the dynamic programming method. However, the premise of using model-based methods is that the current environment is known and well-described. Therefore, the approximation of the model-based method (model-free method) is usually used to deal with the problem. The existing model-free agents are presented in section 2.2.

2.1 Markov decision process

RL is a branch of Machine Learning which tries to maximize the reward it obtains in a complex and uncertain environment. It consists of the agent and the environment. RL trains the agent by constantly interacting with the environment. The agent's purpose is to receive as much reward from the environment as possible through trial and error.

The Markov decision process is one of the most fundamental theoretical models of reinforcement learning, and most problems can be regarded as or transformed into the Markov decision process. A Markov decision process is represented by a 4-tuple $< S, A, P, R >$.

S: Set of all possible states

A: Set of possible actions that the agent can perform in all states.

$P(S \times A \times S \rightarrow [0, 1])$: State transition function represents the probability of taking action to transfer to some state.

$R(S \times A \times S \rightarrow R)$: Reward function represents the reward received from the environment after taking the action which changes the current state.

Fig.1 shows the process of Markov decision. At a time step t, the agent observes the current state $s_t \in S$, and then selects and performs the action from action space A. Then a reward is obtained, the environment moves to a new state, and then the agent continues to repeat the above process until the terminal state or timeout and restarts. The process above can be represented by a trajectory: $s_0, a_0, r_1, s_1, a_1, r_2, s_3....$ a_t means the action performed and r_t denotes the reward when performing a_t in the state s_t. The accumulated return Rt from the time step t with discount factor $\gamma \in (0, 1]$ can be noted as: $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$.

The goal of the agent is to maximize the expected return from each state s_t. The action-value function $Q^{\pi}(s, a)$ is used to estimate the expected return when we take an action in a certain state when following the current policy. The state-value function $V^{\pi}(s)$ is to evaluate the expected return in the current state s when following the π. The action value function or the value function can be eliminated by substitution, and the Bellman expectation equation can be obtained. Similarly, the relationship between the optimal state and the action value function can be defined:

$$\begin{cases} V^*(s) = \max_{a \in A} Q^*(s, a) \\ Q^*(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma V_{\pi}(s')] \end{cases} \tag{1}$$

Then, the Bellman optimal equation can be derived according to the optimal state and the action value function:

$$\begin{cases} V^*(s) = \max_{a \in A}[r(s, a) + \gamma \sum_{s'} p(s'|s, a)V^*(s')] \\ Q^*(s, a) = r(s, a) + \sum_{s'} p(s', r|s, a) \max_{a'} Q^*(s', a') \end{cases} \tag{2}$$

We use the symbol $*$ to indicate the estimated value. $r(s, a)$ indicates the reward when performing a in the state s. We can find an optimal policy through Bellman's expectation and optimal function in theory. However, the Bellman equation is difficult to obtain. Therefore, we need other ways to estimate the optimal value function, such as policy iteration.

2.2 Model-free agent

An optimal policy can be found through Bellman expectation and optimal function in theory. However, the Bellman equation is difficult to obtain. Therefore, we need other ways to estimate the optimal value function, such as policy iteration.

There are three types of model-free agents: Value-Based agent, Policy-Based agent and Actor-Critic agent and Actor-Critic agent combines the first two. Value-based model-free methods learn the value function so that a policy can derive from value function. Policy-based model-free methods directly parameterize the policy by $\pi(a|s;\ \theta)$ and update parameter θ by performing gradient ascent on $E[R_t]$. Actor-Critic agent learns both policy and value function.

A standard policy-based method [9] updates θ by $\nabla_{\theta} R_t \log_{\pi}(a_t|s_t;\ \theta)$, it is an unbiased estimate of $\nabla_{\theta} E[R_t]$. Afterwards, baseline $b_t(s_t)$ from the return has been used in order to reduce the variance of the estimate forward. Then the parameter of policy π is updated by $\nabla_{\theta}(R_t - b_t(s_t))\log_{\pi}(a_t|s_t;\ \theta)$ .The estimated value function can be rewarded as $b_t(s_t)$ to further reduce the variance and $R_t - b_t(s_t)$ can be considered as the advantage of performing the action at in state s_t when we adopt the policy π. As mentioned above, R_t is

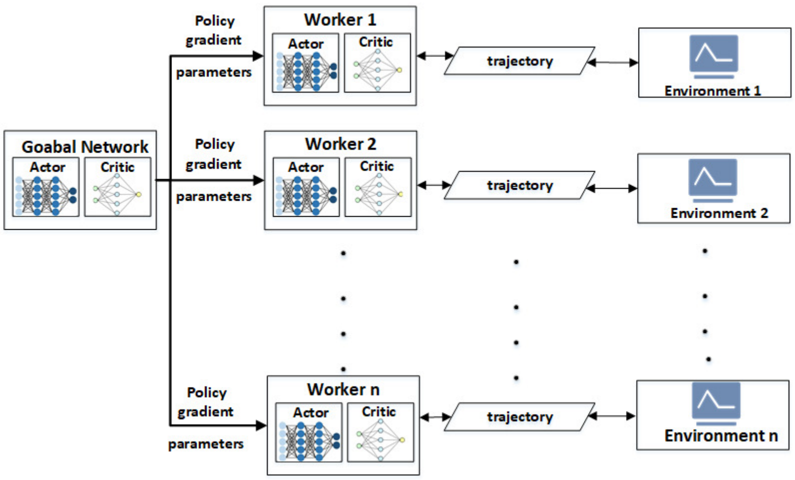

Fig. 2 Mechanism of A2C algorithm

the estimate of the action value function $Q^{\pi}(s_t, a_t)$ and finally the advantage $A(s_t, a_t)$ can be calculated by $Q(s_t, a_t) - V(s_t)$.The method [10, 11] introduced above can be viewed as an Actor-Critic architecture with policy π as an actor and the baseline $V(s_t)$ as a critic.

Advantage Actor-Critic (A2C) [12] is a typical Actor-Critic method, which is a synchronous, deterministic variant of Asynchronous Advantage Actor-Critic (A3C). It uses multiple workers to avoid the use of a replay buffer. Fig.2 shows the mechanism of A2C.

Most of work [13–16] based on RL adopts Q-learning as agent. Q-learning uses a table to record expected values of actions that are called action values in a specific state. Recently, some researchers replace tabular methods with Deep Neural Networks (DNN). DeepGUIT represents the value function by DQN, and ARES adopts TD3, DDPG, and SAC to fit value functions.

3 Approach

This paper proposes a method of Android GUI testing based on A2C and PPO. This section will describe the GUI testing of APP applying reinforcement learning in detail. The process of testing APPs can be thought of as a Markov decision process. A2C and PPO algorithms were used to generate test cases, respectively.

First, Appium extracts specific information about an APP installed on an Android device or emulator. Appium is an open-source automated testing tool for native, mobile, or hybrid APPs. Android devices communicate with Appium through the Android Debug Bridge (ADB). The GUI state composed of GUI elements exists in the form of an XML file that includes information such as package name, executable events (such as whether the widget is clickable, long-clickable, or scrollable), boundary information, and resource-id.

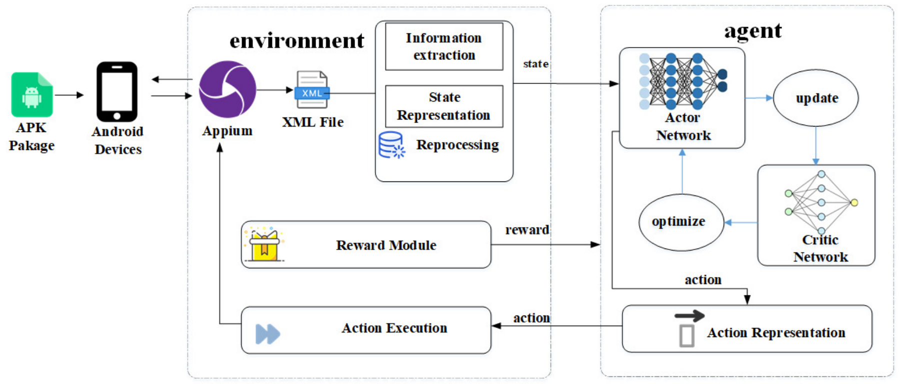

Fig. 3 workflow of ATAC and ATPPO

Then, the XML file is preprocessed. Since the XML file saves the page information, the information is extracted from the page. Then all the executable widgets are obtained. Afterward, the page's state(which can be processed by the neural network) is expressed according to the widget's information. Then the current state can be passed to the agent. According to the current state, the agent makes decisions and outputs actions. The digital action is converted into a specific operation, and the corresponding process is executed. Once the action is performed, the reward module rewards the agent for the action in its current state. The agent adjusts the parameters inside the neural network according to the reward feedback.

At a time step t, the environment gets the current GUI of the Android device through Appium, extracts the state from it, and passes the observation to the agent. Then, the agent (Actor-network follows policy π) predicts an action likely to archive more accumulated rewards. Once the action is performed, the agent will obtain a reward as feedback from the environment. The agent will adjust the actor and critic networks according to the rewards. The workflow is shown in Fig.3.

3.1 Representation of States and Actions

In the reinforcement learning process, the current state needs to be abstractly represented to express the form the neural network can understand. The state representation needs to combine the parts that are beneficial to the training of the agent. The GUI of an Android application consists of GUI elements, and a GUI interface contains many widgets. This method only considers the executable GUI elements in the interface. We only consider the executable GUI elements to represent the states so that they can be passed to and processed by the neural network. We denote the abstract state by all executable widgets from the GUI of the application. Each widget is represented by a three-dimensional vector w_i, and each state is represented by s_t:

$$s_t = (w_1, w_2, w_3, ..., w_i, ...w_n) \tag{3}$$

The first dimension of w_i indicates whether the widget i is clickable, the second dimension indicates whether i is long-clickable, and the third dimension indicates whether the i is scrollable. If the widget i is clickable, long-clickable, or scrollable, the corresponding dimension will be marked as 1. Otherwise, we will mark the dimension as 0. For example, if widget i is clickable and can be long-clickable but not scrollable, we would denote w_i as [1,1,0]. At the time t, the agent observes a state s_t composed of n widgets, where n is the total number of executable widgets in the GUI interface.

Both system-level actions and typical actions are considered in our approach. The system-level actions include switching the internet connection state, screen rotation, and return. There is no return button in many applications explicitly, and the applications may keep in a stalemate state, which prevents them from exploring more space. ATAC adopts a similar action representation as ARES [8]. A 3-dimensional vector denotes each action, and the first dimension specifies which widget or system-level action will be operated. The second dimension works when the widget expects an input since the dimension indicates the index of a string in a predefined dictionary. The third dimension acts as a compliment. The third dimension decides which actions to perform when a widget is clickable and long-clickable. When a widget is scrollable, the third dimension specifies the scrolling direction.

3.2 Reward Function

The design of the reward function is a crucial step where the agent adjusts its strategy based on the feedback (reward) it receives after performing a particular action in the current state (GUI interface). The value of the reward reflects which behaviors are encouraged and which are discouraged. A positive reward value indicates that the current operation is encouraged, and a negative reward value suggests that the current process is discouraged. The larger the reward value, the more expected the action was performed in the current state.

The reward function needs to be designed with the ability to detect faults in mind. However, a few failures in GUI testing can lead to sparse rewards. Intuitively, exploring more GUI space might lead to finding more bugs, so the design of the reward function also considered whether more GUI space exploration could be done. The reward function in this article depends mainly on the change of state and whether an error can be detected. There are two main aspects to the state change process, one is whether a state can be explored that was not explored during the previous test, and the other is whether the executable widget has never been executed in its current state. When an agent chooses to return, there are two aspects to consider. First, we want it to return as soon as the application enters a deadlock state. Second, we don't expect frequent returns, which might interfere with our application exploration. According to the particularity of reward, two kinds of reward functions are defined in this paper.

The definition of reward function is as follows:

a)When the agent performs actions:

$$r_t = \begin{cases} R_1, & crash \\ -R_4, & s_{t+1} \notin AUT \\ -R_3, & stopTime < N \\ R_3, & stopTime \geq N \end{cases} \tag{4}$$

b)When the agent performs other actions:

$$r_t = \begin{cases} R_1, S_{t+1} \notin visitedState \quad or \quad crash \\ R_2, S_{t+1} \in visitedState \quad and \quad a_t \notin visitedWidgets \\ -2^{repeat_{time}}, S_{t+1} \in visitedState \quad and \quad a_t \in visitedWidgets \\ -R_4, S_{t+1} \notin AUT \end{cases} \tag{5}$$

The numbers in the function satisfy the conditions that $R_1 > R_2 > R_4 > R_3.R_1 = 1000.R_2 = 500, R_4 = 100$ and $R_3 = 50$. We encourage this behavior when an application crashes. Either the agent explores a new state or manipulates a widget that hasn't been executed before. *stop_time* indicates how long it is in the current state. A positive reward is given if the application stays in the same state for a long time and the agent performs a return operation. If the agent frequently returns when the application is in a new state, we give it a negative reward. *repeat_time* indicates the number of times the same action is executed consecutively.

3.3 Advantage Actor-Critic(A2C) based Testing

A2C (Advantage Actor-Critic) is a synchronous variation of A3C (Asynchronous Advantage Actor-Critic). They all maintain a network of policies and a network of value functions. They choose the same advantage function $A(a_t, s_t) = (r_{t+1} + \gamma V(s_t) - V(s_t))$ and use multithreading to perform gradient descent. The difference between them is the time to update the global policy and the value function. A2C is synchronous, and A3C is asynchronous. A2C uses multiple threads to avoid using the experience replay buffer.

A2C constructs multiple threads that interact with the environment. In each iteration, the global network waits for the threads to complete their respective turns and updates the global network through the gradient uploaded by the threads. The global network then sends the latest network parameters to all threads simultaneously. The detailed algorithm for each thread is shown in Algorithm 1.

3.4 Proximal Policy Optimization(PPO) based Testing

With the continuous improvement of the reinforcement learning algorithm, the PPO algorithm is proposed to improve the A2C algorithm. This paper also uses the PPO algorithm as the training and decision-making algorithm. The PPO algorithm combines the multi-threading thought of the A2C algorithm and the

Algorithm 1 Advantage Actor-Critic Algorithm for each worker(thread)

```
Require: global actor network $\theta_\pi$, global critic network $\theta_v$
Ensure: accumulated actor network gradient $d\theta_\pi$, critic network gradient $d\theta_v$
 1: repeat
 2:     Reset gradients:$d\theta_\pi \leftarrow 0$, $d\theta_v \leftarrow 0$
 3:     Observe the GUI state $S_t$
 4:     repeat
 5:         $a_t \leftarrow \pi\ (a_t|s_t,{\theta_\pi}')$
 6:         Perform an action $a_t$ and Receive a reward $r_t$
 7:         Observe a new GUI state $s_{t+1}$
 8:     until $s_t$ is terminal or timeout
 9:     for each step i of episode do
10:         $R \leftarrow r_i + \gamma R$
11:         $d\theta_\pi \leftarrow d\theta_\pi + \nabla_{\theta'_\pi} \log\pi(a_i|s_i;\ \theta'_\pi)(R - V(s_i;\ \theta'_\pi))$
12:         $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\ \theta'_\pi))^2/\partial\theta'_\pi$
13:     end for
14: until timeout
15: return $d\theta_\pi, d\theta_v$
```

idea of using the trust domain to enhance the actor of the TRPO algorithm. Both A2C and PPO algorithms are reinforcement learning methods based solely on strategy from time to time. In essence, they are both action-based and evaluator methods, including value function and strategy function.

Reinforcement learning can use on-policy learning and off-policy learning. On-policy reinforcement learning indicates that agents need to interact with the environment during the learning process. On-policy reinforcement learning trains the models by the current strategy, and each piece of data is only used once. PPO adopts off-policy reinforcement learning. Standard policy-based methods perform a gradient update for each data sample. PPO reuses the data for the multi-stage minor batch update. The main idea of PPO is that after updating a new policy, it should be similar to the previous policy, so the scope of parameter updating should be limited. Using importance sampling, the model of this stage can be updated through the data of strategies of different scenes. This is done by adding the TD-Error of other actions to the probability of action and multiplying it by the gradient of the feedback of different strategies.

3.5 Construction and Application of FSM

When exploring the reinforcement learning strategy, some operations need to be executed continuously to trigger a function or function. For example, the user login function must first enter the user name, enter the password, and click the confirm button to realize the login function. However, each step may be interrupted. The longer the sequence, the higher the probability of interruption, which makes it more challenging to achieve the ideal transition,

Algorithm 2 Proximal Policy Optimization Algorithm

```
Require: initial policy $\theta$
Ensure: up-to-date policy $\theta$
1: for iteration=1,2,... do
2:     for iteration=1,2,...,N do
3:         Use policies $\pi_{\theta_{old}}$to Interact with the environment within time T
4:         Calculate advantage to estimate $A_1, .., A_T$
5:     end for
6:     Optimize $\theta$ according to K epochs and small batch data minibatch
7:     $\theta_{old} \leftarrow \theta$
8: end for
9: return $\theta$
```

especially when facing a long path. FSM M can be defined as a five-tuple (S, A, δ, s_0, F), in which S is a finite set of states, A is a set of actions, $\delta : S \times A \rightarrow S$ is a set of transitions, $s_0$is an initial state, and F is a termination state that cannot be transferred to other states. During the testing process, the FSM will update continuously.

When the algorithm may enter a local optimal state, we need to find a path to reach a new state if we want to resume exploration. FSM contains essential information such as state, action, and transition, which can guide subsequent selection. We chose the least visited state in the least explored as the starting point to explore Android APP again. We use the Floyd algorithm to identify the shortest path that can reach the state and then executes the corresponding operation sequence to guide to reach the target state and execute the conversion.

Floyd algorithm is based on greedy and dynamic programming and is similar to the Dijkstra algorithm. Dijkstra algorithm applies to the solution of the shortest path of a single source, and Floyd algorithm applies to the search of the shortest path between multiple source points in the weighted graph.Floyd algorithm contains two matrices, the shortest distance matrix $D_{n\times n}$ of a graph and the shortest path matrix $path_{n\times n}$. The element $d_{i,j}$ n, the shortest distance of the graph, represents the distance from node i to node j. In using reinforcement learning to test Android APP GUI, nodes represent various states. At the initial stage of the test, the distance is positive and infinite. Each time a state transition occurs, the value in the shortest distance matrix may change because the distance between two states will be updated to the shortest distance. If the distance between two nodes is shortened, the shortest path will also change. The algorithm updates the shortest distance matrix and path according to the state transition equation.

The target state can be easily reached when the shortest path matrix is solved. If the state s_i wants to transfer to the state s_j, it needs to move to the state $s_{path_{i,j}}$ first, then from the state $s_{path_{i,j}}$ to the target state s_j. With the state transition process, the target state can be reached according to the shortest path between states, and further exploration can be carried out on this basis.

4 Empirical Study

When testing the application, we want to be able to detect exceptions. In theory, it is only possible to detect more anomalies by exploring as many states as possible. In many studies [7, 8, 14, 16, 17], code coverage and fault detection number are used as evaluation indicators. Therefore, this chapter mainly investigates the code coverage capability and fault detection capability of the proposed method. We evaluated the following metrics: instruction coverage, branch coverage, line coverage, method coverage, and number of failures. ATAC and ATPPO are compared with the state-of-art tools, Monkey, and ARES. Our empirical study is designed to answer the following research questions:
RQ1: Do ATAC and ATPPO achieve higher instruction coverage, branch coverage, line coverage, and method coverage than state-of-art testing tools?
RQ2: Can ATAC and ATPPO reveal more failures than state-of-art testing tools?
RQ3: Can the introduction of FSM into the reinforcement learning framework prevent the algorithm from entering the local optimal state and trigger more functions?

4.1 Applications under Test

Although this approach is black-box, in order to compare with other approaches, you must obtain the source code to gather coverage information, and the experiment uses the Jacoco [18] plug-in to generate coverage information. The experiment selected 20 open source F-Droid applications from Github for method evaluation. The applications selected for the experiment are also used by other researchers studying mobile application GUI testing, and the experiment excluded many applications that were outdated or no longer operational. Some of the applications selected for the experiment are still being modified and iterated. Details of the application under test are shown in Table 1. To facilitate the calculation of the coverage information of the application, the apk package and source code of the AUT have been uploaded to Github [1].

4.2 Evaluation Setup

We compare the proposed methods with the tools Monkey and ARES. Monkey is a random test generation tool, and ARES is based on deep reinforcement learning. Then, specific experimental settings are introduced.

We experiment on a computer with the MacOS operating system, M1 processor, and 16GB of memory. Since the simulator is unstable, the application in the experiment runs on an actual Android device, the operating system is

[1] https://github.com/RL-ATAC/AUT-of-ATAC

Table 1 Target applications for evaluation

Applications	instructions	braches	lines	methods	classes
QuickSettings	470	28	115	17	4
MunchLife	954	63	214	34	12
Silent-ping-sms	1,732	137	324	39	7
BatteryDog	2,945	164	600	72	18
AnyCut	1,527	97	379	72	18
Afwall	63,182	5,397	14,004	2,155	337
Jamendo	18,382	1,103	4,467	909	205
microMathematics	126,031	13,100	24,636	3,262	486
AnyMemo	50,311	2,732	11,251	2,476	547
AmazeFileManager	85,537	7,418	20,471	3,018	488
zooborns	3,543	259	780	132	24
AntennaPod	59,826	4,909	14,830	2,587	385
BudgetWatch	5,248	319	1,194	168	51
Drawablenotepad	2,498	132	563	124	22
MutiSmsSender	3,736	235	826	126	32
RedReader	102,238	8,427	23,748	3,731	657
BirthdayCountDown	1,607	68	262	30	5
materialistic	30,745	2,664	7,227	1,815	274
BMI	510	45	103	25	5
LockPattern	2,816	217	642	128	28

Android 10, and the memory is 6GB. The framework of ATAC is based on ARES. It realizes the A2C algorithm mainly by OpenAI Gym [19]. The communication between applications and agent is through Appium [20], and Android Debug Bridge [21]. The structures of the Actor and Critic networks are the same, and ATAC adopts two 2-layers neural networks. There are 64 neurons in each layer. ATAC and ATPPO choose relu as the activation function in both neural networks. The number of actions that can be performed changes from time to time in each state, so the Gaussian distribution is adopted as output, representing the probability distribution of actions in the current state. The ATAC and ATPPO parameters are shown in Table 2 and 3, respectively.

The experiment also builds FSM during ARES and ATAC testing, respectively. When the experiment may be in the local optimal state, we find the least frequently visited state and the shortest path to the state in transition through Floyd algorithm under the guidance of FSM. By performing the actions in the shortest route, reaching the target state, and continuing exploring. The condition to determine whether the local optimal state may be achieved is if the new state can be found in 30 steps or if it stays on the current page after repeating 30 steps.

The length of an episode is set as 250. The agent explores and tests the application within 5000-time steps, limiting the time to one hour. The same settings are used in ARES. The throttle setting was 200 in Monkey, and the procedure lasted one hour. The tool was configured to ignore any crash, system timeout, and security exceptions until the timeout was reached. The running log of the application is analyzed to obtain the crash information. Reinforcement learning has a certain randomness. We repeated all the experiments five

Table 2 Specific parameter settings of ATAC

Parameter name	Parameter value	Parameter name	Parameter value
learning_rate	0.0007	use_rms_prop	True
n_steps	5	max_grad_norm	0.5
gamma	0.99	total_timesteps	5000
ent_coef	0	n_eval_episodes	5
vf_coef	0.5	rms_prop_eps	1e-05

Table 3 Specific parameter settings of ATPPO

Parameter name	Parameter value	Parameter name	Parameter value
learning_rate	0.0003	gamma	0.99
n_steps	2048	gae_lamda	0.95
batch_size	64	clip_range	0.2
n_epochs	10	ent_coef	0
vf_coef	0.5	normalize_advantage	True

times and used the average value of 5 times to represent the final result to avoid the randomness of the results. The comparative experiment was also repeated five times to take the average value.

4.3 RQ1:Do ATAC and ATPPO achieve higher code coverage than state-of-art testing tools?

To answer RQ1 and avoid the randomness of the results as much as possible, we repeat all the experiments five times and use the average of five executions to represent the final result, including the comparative experiment. Table 4 and Table 5 show the average instruction coverage of ATAC and ATPPO. Inst represents instruction coverage, bran represents branch coverage, the line represents line coverage, and meth means method coverage.

Table 3 shows the results of comparing ATAC with the coverage of ARES and Monkey. It can be seen that ATAC covers more instructions, branches, lines, and methods on 20 APPs. It can be noted that the average instruction coverage (50.8%) of ATAC is higher than Monkey (35.0%) and ARES (47.5%). The average branch coverage (36.5%) is also higher than Monkey (21.3%) and ARES (33.1%). The average line coverage (51.2%) is also higher than Monkey (34.1%) and ARES (47.9%), and the average method coverage (55.4%) is also higher than Monkey (40.3%) and ARES (52.1%).

Regarding average instruction coverage and branch coverage, ATAC performs best in 16 of the 20 APPs, ARES performs best in 4, and Monkey performs poorly. In terms of average line coverage, Monkey also has no outstanding performance. ARES performs best on 4 APPs, and ATAC performs best on 16 APPs. Regarding average method coverage, ATAC is the best on most apps, ARES is the best on three apps, and Monkey is the best on only

Table 4 Coverage of ATAC(%)

apps under test	ATAC				ARES				Monkey			
	inst	bran	line	meth	inst	bran	line	meth	inst	bran	line	meth
QuickSettings	**88.6**	**72.4**	**87.7**	82.8	84.6	63.8	84	81.4	72.9	45.0	72.6	**86.8**
MunchLife	**83.6**	**59.0**	**84.0**	**88.2**	81.2	51.8	80.8	88.2	59.0	33.8	53.5	66.4
Silent-ping-sms	**43.0**	**27.0**	**47.2**	**59.0**	42.0	23.0	46.3	59.0	30.8	11.5	30.3	42.4
BatteryDog	**71.0**	**55.0**	**69.8**	**73.6**	69.0	52.6	68.4	70.8	65.4	50.0	61.5	66.1
AnyCut	**68.6**	**52.4**	**69.5**	**75.0**	68.0	49.2	68.9	72.3	39.4	22.2	39.8	50.1
Afwall	**14.8**	**11.4**	**15.5**	**19.1**	12.8	9.6	13.4	17.5	6.0	4.0	7.0	10.0
Jamendo	**23.4**	**14.4**	**23.4**	**28.5**	16.7	9.0	16.3	22.3	10.9	6.5	11.6	14.9
microMathematics	37.2	25.2	36.0	44.5	**42.4**	**29.4**	**41.2**	**48.8**	24.4	15.0	24.3	31.6
AnyMemo	**37.8**	**26.4**	**38.6**	**43.0**	24.6	16.3	25.7	28.9	17.7	11.0	19.3	23.3
AmazeFileManager	**28.8**	**19.8**	**27.5**	**35.9**	16.0	9.2	15.6	22.6	15.4	8.9	14.6	20.6
zooborns	**18.2**	**7.4**	**19.8**	**24.5**	17.4	7.6	18.6	24.3	11.5	4.8	12.5	17.1
AntennaPod	**28.0**	**17.8**	**26.4**	**28.2**	23.2	13.6	22.7	23.5	10.4	5.6	9.2	9.8
BudgetWatch	**41.0**	**27.8**	**44.3**	**56.4**	39.8	26.2	43.0	55.4	19.8	9.1	20.9	25.9
Drawablenotepad	**87.8**	**76.3**	**86.4**	**85.3**	83.8	70.8	82.7	81.8	57.4	30.5	56.3	60.7
RedReader	**38.0**	**26.8**	**35.6**	**42.9**	27.4	22.0	27.4	33.0	14.4	8.5	14.1	18.5
BirthdayCountDown	90.8	62.2	90.9	90.0	**92.0**	**65.0**	**92.0**	**91.3**	78.4	47.6	73.1	78.0
materialistic	**51.4**	**32.6**	**51.6**	**55.2**	46.2	27.8	46.4	50.1	28.2	15.2	28.5	32.4
BMI	62.0	30.0	66.0	80.0	**67.0**	**38.6**	**68.5**	**77.6**	62.0	36.0	60.8	80.8
LockPattern	64.0	54.0	61.7	54.7	**64.8**	**54.4**	**62.4**	**55.9**	60.4	53.2	57.1	51.9
MutiSmsSender	**38.6**	**31.2**	**41.5**	**41.1**	31.4	21.8	33.4	37.0	15.4	8.2	15.8	18.1
AVERAGE	**50.8**	**36.5**	**51.2**	**55.4**	47.5	33.1	47.9	52.1	35.0	21.3	34.1	40.3

Table 5 Coverage of ATPPO(%)

apps under test	ATAC				ARES				Monkey			
	inst	bran	line	meth	inst	bran	line	meth	inst	bran	line	meth
QuickSettings	**90.0**	**71.0**	**89.6**	82.4	84.6	63.8	84	81.4	72.9	45.0	72.6	**86.8**
MunchLife	**82.6**	**57.6**	**82.9**	**88.2**	81.2	51.8	80.8	88.2	59.0	33.8	53.5	66.4
Silent-ping-sms	**43.0**	**27.0**	**47.2**	**59.0**	42.0	23.0	46.3	59.0	30.8	11.5	30.3	42.4
BatteryDog	**71.0**	**55.0**	**70.0**	**73.6**	69.0	52.6	68.4	70.8	65.4	50.0	61.5	66.1
AnyCut	**69.0**	**53.0**	**69.7**	**75.0**	68.0	49.2	68.9	72.3	39.4	22.2	39.8	50.1
Afwall	**14.4**	**11.2**	**14.0**	**16.9**	12.8	9.6	13.4	17.5	6.0	4.0	7.0	10.0
Jamendo	**23.0**	**14.0**	**23.3**	**28.6**	16.7	9.0	16.3	22.3	10.9	6.5	11.6	14.9
microMathematics	37.2	25.0	36.1	44.3	**42.4**	**29.4**	**41.2**	**8.8**	24.4	15.0	24.3	31.6
AnyMemo	**49.2**	**34.6**	**50.1**	**54.9**	24.6	16.3	25.7	28.9	17.7	11.0	19.3	23.3
AmazeFileManager	**29.8**	**20.6**	**28.2**	**36.1**	16.0	9.2	15.6	22.6	15.4	8.9	14.6	20.6
zooborns	**17.6**	**7.6**	**19.6**	**24.4**	17.4	7.6	18.6	24.3	11.5	4.8	12.5	17.1
AntennaPod	21.6	12.4	20.4	24.1	**23.2**	**13.6**	**22.7**	**23.5**	10.4	5.6	9.2	9.8
BudgetWatch	**40.2**	**28.4**	**44.1**	55.1	39.8	26.2	43.0	**55.4**	19.8	9.1	20.9	25.9
Drawablenotepad	**84.4**	**69.8**	**83.7**	**83.1**	83.8	70.8	82.7	81.8	57.4	30.5	56.3	60.7
RedReader	**40.8**	**28.8**	**40.7**	**46.8**	27.4	22.0	27.4	33.0	14.4	8.5	14.1	18.5
BirthdayCountDown	84.2	52.6	83.7	84.0	**92.0**	**65.0**	**92.0**	**91.3**	78.4	47.6	73.1	78.0
materialistic	**51.6**	**32.2**	**51.7**	**56.1**	46.2	27.8	46.4	50.1	28.2	15.2	28.5	32.4
BMI	62.6	31.4	66.2	80.0	**67.0**	**38.6**	**68.5**	77.6	62.0	36.0	60.8	**80.8**
LockPattern	63.4	53.4	61.1	54.7	**64.8**	**54.4**	**62.4**	**55.9**	60.4	53.2	57.1	51.9
MutiSmsSender	**37.4**	**32.2**	**39.7**	**35.5**	31.4	21.8	33.4	37.0	15.4	8.2	15.8	18.1
AVERAGE	**50.7**	**35.9**	**51.1**	**55.1**	47.5	33.1	47.9	52.1	35.0	21.3	34.1	40.3

two apps. By applying the A2C algorithm, ATAC achieves higher instruction, branch, line, and method coverage than state-of-art tools Monkey and ARES.

The results of the code coverage of ATPPO, ARES, and Monkey are shown in Table 5. By analyzing the coverage results, we can find that the average instruction coverage of ATPPO(50.7%) was higher than Monkey(35.0%) and ARES(47.5%). The average branch coverage of ATPPO(35.9%) was also higher than Monkey(21.3%) and ARES(33.1%). The average line coverage (51.1%)

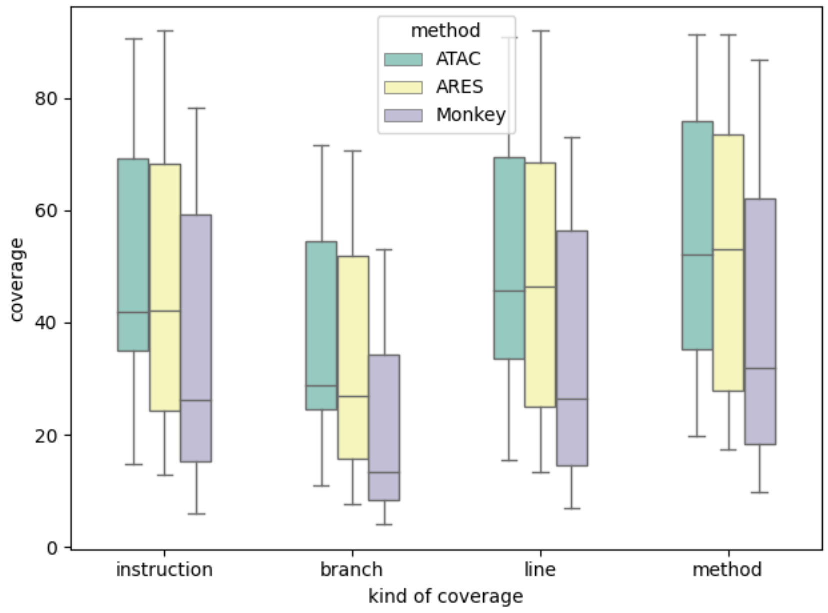

Fig. 4 Code Coverage achieved by Monkey, ARES and ATAC

was also higher than that of Monkey(34.1%) and ARES(47.9%), and the average method coverage (55.1%) was higher than that of Monkey(40.3%) and ARES(52.1%). ATPPO achieves higher instruction, branch, line, and method coverage in 15 of 20 apps.

ATPPO performed best in 15 out of 20 applications, ARES performed best in 5 out of 20 applications, Monkey performed poorly and only performed best in QuickSettings in method coverage. For some reason, Monkey doesn't perform well on data sets running on Android 10.

In short, ATPPO achieves higher instruction coverage, branch coverage, line coverage, and method coverage than the existing advanced methods Monkey and ARES tools. ATAC and ATPPO perform better than ARES and Monkey regarding code coverage. ATAC performs better than ATPPO in the number of apps and code coverage.

4.4 RQ2: Can ATAC and ATPPO reveal more failures than state-of-art testing tools?

We explore five times on each APP. This method may find the same exception multiple times in each iteration. ATAC finds exceptions in 8 out of 20 apps, ATPPO also finds exceptions in 7 apps, ARES finds exceptions in 4 apps, and Monkey finds exceptions in 2 apps.

Since no exceptions are found in the other 11 apps, this section only focuses on the nine apps in which the method proposed in this paper, ARES and Monkey found exceptions. ARES and Monkey can't find exceptions that ATAC and ATPPO find in Jamendo, AmazeFileManager, AnyMemo, AntennaPod,

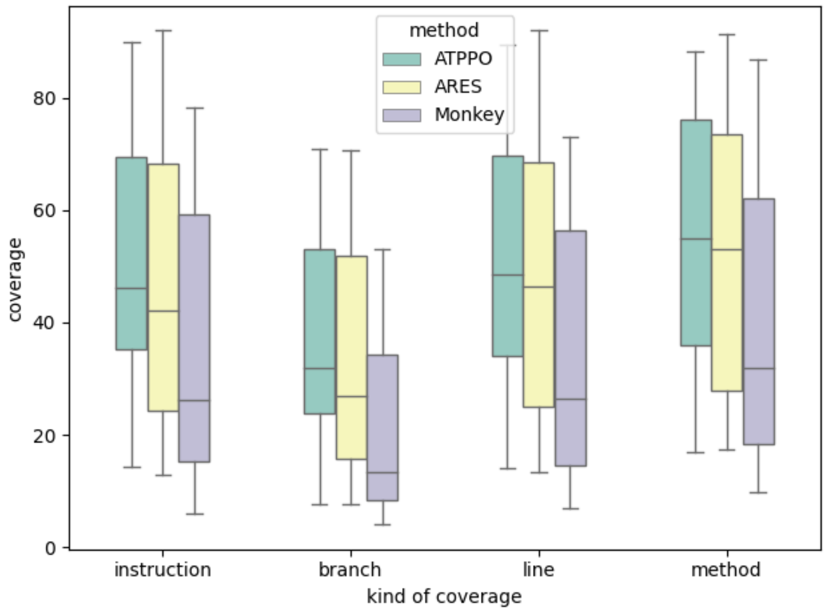

Fig. 5 Code Coverage achieved by Monkey, ARES and ATPPO

Table 6 The number of experiments where failures is found

Applications	ATAC	ATPPO	ARES	Monkey
Jamendo	2	4	0	0
AmazeFileManager	1	2	0	0
AnyMemo	3	3	0	0
Drawablenotepad	3	5	1	0
zooborns	5	5	5	1
AntennaPod	1	0	0	0
BatteryDog	5	5	5	1
materialistic	3	0	0	0
MutiSmsSender	0	1	1	0

and materialistic. Table 6 records the number of experiments where failures are seen five times. TATAC and ATPPO have the best performance and can find exceptions on more apps. All the methods detect some exceptions in the five experiments in zooborns and BatteryDog.

Table 6 and Table 7 show that ATAC and ATPPO can reveal as many exceptions as possible, and ATPPO finds more exceptions than other methods. Since many of the exceptions discovered by researchers have been fixed, and the source code of most apps is still being iterated and modified, the number of exceptions that can be found decreases with each release. Exceptions found by ATAC and ATPPO include RuntimeException, NullPointerException, and NumberFormatException. Table 8 describes the detailed exceptions.

Table 7 The kinds of experiments where failures is found

Applications	ATAC	ATPPO	ARES	Monkey
Jamendo	1	1	0	0
AmazeFileManager	1	3	0	0
AnyMemo	2	4	0	0
Drawablenotepad	1	2	1	0
zooborns	1	1	1	1
AntennaPod	1	0	0	0
BatteryDog	1	1	1	1
materialistic	3	0	0	0
MutiSmsSender	0	1	1	0

Table 8 The number of experiments where failures is found

Applications	descriptions of exceptions
Jamendo	An error occurred while executing doInBackground()
AmazeFileManager	IndexOutOfBoundsException
AmazeFileManager	Unable to start activity ComponentInfocom.android.certinstaller/com.android.certinstaller.CertInstallerMain
AmazeFileManager	InvocationTargetException
AnyMemo	Unable to start activity ComponentInfo
AnyMemo	InvocationTargetException
AnyMemo	IndexOutOfBoundsException
AnyMemo	Attempt to invoke virtual method 'java.lang.Integer org.liberty.android.fantastischmemo.entity.Card.getId()' on a null object
AnyMemo	start failed
AnyMemo	For input string: "string2"
Drawablenotepad	InvocationTargetException
Drawablenotepad	IndexOutOfBoundsException
zooborns	Attempt to get length of null array
AntennaPod	For input string: "string2"
BatteryDog	Attempt to invoke interface method 'void android.view.Menu.clear()' on a null object reference
materialistic	For input string: "string6"
materialistic	Unable to start activity ComponentInfocom.android.certinstaller/com.android.certinstaller.CertInstallerMain
materialistic	Attempt to invoke virtual method 'java.lang.Integer org.liberty.android.fantastischmemo.entity.Card.getId()' on a null object reference
MutiSmsSender	Unable to start activity ComponentInfocom.hectorone.multismssender/com.hectorone.multismssender.PhoneNumberSelection

Compared to ARES and Monkey, ATAC performs best on 16/20 apps, while ATPPO performs best on 15/20 apps regarding code coverage. ATAC and ATPPO can find more exceptions in terms of exception detection. At the same time, ATPPO finds the most types of exceptions. Experiments have shown that ATAC and ATPPO can achieve higher code coverage and find more number and variety of exceptions than ARES and Monkey.

4.5 RQ3: Can the introduction of FSM into the reinforcement learning trigger more functions?

To answer RQ3, we established FSM in the testing process based on ARES and ATAC, respectively. When the experiment is in the local optimal state, the action is performed under the guide of FSM. Branch coverage and method coverage are used to evaluate the degree of exploration of each APP. The average branch coverage and average method coverage of each method in five experiments are calculated, where branch represents branch coverage and method means method coverage.

Table 9 compares the branch coverage and method coverage of ARES methods with and without FSM guidance on 20 apps. In the app data set of this

Table 9 Branches and method coverage of ARES and FSM-guided ARES (%)

apps under test	FSM-guide ARES		ARES	
	bran	meth	bran	meth
QuickSettings	54.4	**82.4**	**63.8**	81.4
MunchLife	**59.8**	**88.2**	51.8	88.2
Silent-ping-sms	**27.0**	**59.0**	23.0	59.0
BatteryDog	**55.2**	**73.3**	52.6	70.8
AnyCut	**50.4**	**75.0**	49.2	72.3
Afwall	**12.4**	**19.8**	9.6	17.5
Jamendo	**12.2**	**26.3**	9.0	22.3
microMathematics	28.0	48.6	**29.4**	**48.8**
AnyMemo	**27.0**	**44.3**	16.3	28.9
AmazeFileManager	**22.6**	**35.7**	9.2	22.6
zooborns	**11.0**	**33.3**	7.6	24.3
AntennaPod	**18.8**	**32.7**	13.6	23.5
BudgetWatch	**28.4**	**56.0**	26.2	55.4
Drawablenotepad	**71.6**	**83.1**	70.8	81.8
RedReader	**47.0**	**46.6**	22.0	33.0
BirthdayCountDown	**66.6**	**91.3**	65.0	91.3
materialistic	25.2	45.3	**27.8**	**50.1**
BMI	30.8	**79.2**	**38.6**	77.6
LockPattern	**54.6**	**55.9**	54.4	55.9
MutiSmsSender	**29.2**	34.1	21.8	**37.0**
AVERAGE	**36.6**	**55.5**	33.1	52.1

experiment, the average branch and method coverage of ARES with FSM guidance is higher than that of ARES without FSM guidance. The average branch coverage of ARES with FSM guidance is 36.6%, and the method coverage is 55.5%. Without FSM guidance, the branch coverage of ARES is 33.1%, and the method coverage is 52.1%.

The ARES method with FSM guidance performed best in 16 of 20 applications in the branch coverage, while the ARES method without FSM guidance performed best in only 4 of them. In terms of method coverage, ARES method and FSM guidance have improved the effect of 13 APPs. The method coverage of the two models on MunchLife, Silent ping sms, BirthdayCountDown, and LockPattern is the same, and the ARES method without FSM guidance on the three applications performs better. In general, using FSM to guide ARES improves the coverage of branches and methods on most apps, provides the possibility of triggering more functional methods or more functions, offers advanced guidance for the execution of long specific sequences, and alleviates the possible local optimization problems.

Table 10 shows the branch coverage and method coverage of ATAC, including FSM guidance and excluding FSM. On 20 APP, the ratio of ATAC with FSM guidance is higher than without guidance. The average branch coverage of the A2C-based GUI test model with FSM guidance is 37.7%, and the method coverage is 56.9%, while the branch coverage of the ATAC without guidance is 36.5% and the method coverage is 55.4%. Branch coverage and method coverage are analyzed, respectively. Regarding branch coverage, ATAC

Table 10 Branches and method coverage of ATAC and FSM-guided ATAC(%)

apps under test	FSM-guide ATAC		ATAC	
	bran	meth	bran	meth
QuickSettings	**72.4**	**82.8**	72.4	82.8
MunchLife	**59.8**	**88.2**	59.0	88.2
Silent-ping-sms	23.0	**59.0**	**27.0**	59.0
BatteryDog	53.0	71.6	**55.0**	**73.6**
AnyCut	**53.0**	**75.0**	52.4	75.0
Afwall	**17.6**	**27.8**	11.4	19.1
Jamendo	14.0	28.1	**14.4**	**28.5**
microMathematics	**31.0**	**50.0**	25.2	44.5
AnyMemo	**27.2**	**43.2**	26.4	43.0
AmazeFileManager	**21.4**	**39.8**	19.8	35.9
zooborns	**11.0**	**27.3**	7.4	24.5
AntennaPod	**18.4**	**29.6**	17.8	28.2
BudgetWatch	**32.0**	**62.5**	27.8	56.4
Drawablenotepad	**77.0**	**86.0**	76.3	85.3
RedReader	**30.0**	**48.9**	26.8	42.9
BirthdayCountDown	**64.0**	**90.0**	62.2	90.0
materialistic	**33.2**	**55.2**	32.6	55.2
BMI	**34.0**	**80.0**	30.0	80.0
LockPattern	**54.0**	**54.7**	54.0	54.7
MutiSmsSender	**32.5**	**45.2**	31.2	41.1
AVERAGE	**37.7**	**56.9**	36.5	55.4

with FSM guidance is the best, while ATAC without guidance is only the best on silent ping SMS, BatteryDog, and Jamendo. In terms of method coverage, FSM has instructed ATAC to improve its effect on 10 APPs. The method coverage of these two algorithms is the same on QuickSettings, MunchLife, Silent ping sms, AnyCut, BirthdayCountDown, Materialism, BMI, and LockPattern. The impact of ATAC on BatteryDog and MutiSmsSender without guidance is not apparent at a young age, but it performs better on large apps. Generally, using FSM-guided ATAC can improve branch and method coverage.

A specific sequence of operations triggers a function. The introduction of FSM into the reinforcement learning framework makes it possible to explore the long operation sequence, which provides the possibility of realizing the ideal transition, triggering and detecting more branches and functions while preventing the algorithm from entering the local optimal state, providing higher guidance for the exploration of the reinforcement learning algorithm.

5 Threats to validity

Internal Threats. The threat to internal validity is the non-deterministic characteristic of reinforcement learning. In different iteration cycles, code coverage and error detection quantity may be different. Each method was performed five times on each application to reduce the threat, and the results of the five experiments were combined to evaluate and analyze the technique. Reinforcement learning-based mobile application GUI testing is black-box, based on the front-end page to find exceptions, and may not cover the back-end logic errors.

External Threats. Limits on the number of apps used for evaluation. Although there are tons of apps, we only tested 20 apps. Experiment with selecting different categories and sizes of applications to reduce this threat. The parameters of the reinforcement learning algorithm are set. Many hyper-parameters are selected according to domain knowledge, and some may have better choices.

6 Related work

6.1 Random Exploration Strategy

In the simplest form, the random strategy only produces UI events, is inefficient at generating system-level events, and can only react to a few occasions in a given situation. One of the advantages of the random exploration strategy is that it can quickly generate UI events, which is suitable for stress testing. However, the disadvantage of this type of approach is that it is difficult to produce highly compatible test cases, resulting in many redundant test cases. Monkey is a tool provided by Google for stability and stress testing. It simulates user-generated events or system events (such as clicking, randomly entering text, and so on). It implements the most basic random strategy, where the user specifies the number of events to be generated, and when that number is reached, the test stops. However, it can create a lot of invalid tests, which makes no sense in testing the application.

Dynodroid [22] is also based on a random exploration strategy and can generate both system events and UI events. It can create system events by examining APP-related events. A novel random algorithm is used to select a widget, either by choosing the least frequently selected event or by taking context into account, allowing the user to provide input manually when exploration stalls, for example, by authenticating the user. Some researchers [23, 24] have introduced fuzzy testing into test applications to generate fuzzy input. These tools are designed to create invalid input, crash the APP, and test the robustness of the test application. Such methods are also quite effective in revealing security vulnerabilities (such as denial of service), highly targeted, and less effective in detecting defects. Intent Fuzzer [23] combines static analysis with random test generation. Droidfuzzer [25] analyzes Intent-filter tags in the AndroidManifest.xml file to target activities.

6.2 Model-based Exploration Strategy

Some mobile application testing methods begin by building a GUI model of the application and then generating events based on that to explore the mobile application. Typically these test methods use finite state automata as a model, abstracting an activity into a state and events as transitions between states.

Based on the depth-first traversal strategy, the GUIRipper [2] dynamically builds the APP model, adding a list of events every time a new state is found. To increase the application coverage, SwiftHand [6] will also need to optimize the exploration strategy by generating a finite state automaton model of the application, which cannot create system events, only simple UI events such as clicking and scrolling. The framework PUMA [26] is easy to extend. It implements the same essential random exploration as Monkey and can also be extended and dynamically analyzed based on the basic exploration strategy. As a model-based approach, PUMA also includes finite-state machines. The framework supports different exploration methods and state representation. However, it has some problems, such as incompatibility with some versions of the framework. Stoat [4] used a random finite state automaton model to describe the behavior of APP. In model construction, the executable events are judged, and their execution priority is determined according to the event type and execution frequency.

6.3 System Exploration Strategy

Functions cannot be triggered by simple clicking, scrolling, or system-level events. They need to be triggered by specific inputs. Some mobile application testing tools use evolutionary or symbolic execution algorithms to explore mobile applications systematically. Although the system exploration strategy can cover the function or function well, it needs to improve in terms of scalability.

EvoDroid [27] uses evolutionary algorithms to systematically explore applications, including fitness functions, to achieve maximum coverage. ACTEve [28] is a symbol testing tool that tracks events from the point in the framework where they are generated to the point where they are processed in the APP. For this reason, ACTEve needs to examine the APP and its framework. ACTEve supports both system and UI events. Sapienz [29] used Pareto-based optimal multi-objective search to maximize code coverage, reveal errors and minimize the length of test sequences. To generate the article field for a specific input, reverse design the APK to get a statically defined string. However, generating new test cases through random crossover and mutation results in generating invalid sequences, and iterative evaluation of newly generated test cases also takes a significant amount of time.

6.4 Machine Learning based Strategy

Machine learning techniques include supervised Learning, reinforcement learning, and Active Learning. Methods based on machine learning have been widely used in the field of testing. Reinforcement learning also does not need to label data sets. Under the guidance of the reward function, the model is trained through trial and error exploration to find the most favorable actions under the current situation.

SwiftHand [30] applies machine learning to learn a model of the app during testing and uses the model to generate user inputs that visit unexplored states of the app. Then, it uses the app's execution on the generated inputs to refine the model. More and more researchers [9–13, 15] try to adopt reinforcement learning into GUI test generation. Most of them [2, 16] utilize Q-learning as an agent which maintains a Q-table to record Q value. Q-testing [14] uses Q-learning and divides different states at the granularity of functional scenarios to efficiently explore other functionalities. The Q-table needs wonderful memories if the states and actions space is enormous. [7, 17] replace Q-learning with Deep Q Network, which utilizes Q network to predict actions in certain states. ARES [8] is a Deep RL approach for black-box testing of Android applications, and it employs DDPG, SAC, and TD3 algorithms as the agents. However, it needs a big replay buffer to save the experience.

7 Conclusion

Applications are constantly updated, and their functions and pages are expanded continuously. They are becoming more and more complex and limited by human resources and time and space, making it more challenging to test apps. Aiming at the problem of state combination explosion and exploring the spatiotemporal limitations of an APP, the paper proposes ATAC and ATPPO. Our approaches apply a deep neural network as an agent and are evaluated in 20 apps and perform better than Monkey and ARES. They have higher instruction, branch, line, and method coverage. The performance of detecting failures is also better. This paper also introduces Finite-State Machine into the reinforcement learning framework to avoid falling into the local optimal state, which provides high-level guidance for further improving the test efficiency. We will compare them with more tools and explore more apps in future work.

8 Declarations

Ethical Approval Not applicable.
Competing Interests The authors declare no conflicts of interest.
Consent to Participate Informed consent was obtained from all individual participants.
Authors' Contributions Chuanqi Tao and Hongjing Guo wrote the main manuscript text, Hongjing Guo and Jerry Gao prepared figures 1-5 and Table 1-10. All authors reviewed the manuscript.
Funding Not applicable.
Availability of Data and Materials The AUT can be reached in https://github.com/RL-ATAC/AUT-of-ATAC.

References

[1] Google: UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey

[2] Amalfitano, D., Fasolino, A.R., Tramontana, P., Carmine, S.D., Memon, A.M.: Using GUI ripping for automated testing of android applications. In: Goedicke, M., Menzies, T., Saeki, M. (eds.) Proceedings of the 12th IEEE/ACM International Conference on Automated Software Engineering, pp. 258–261. ACM, Essen, Germany (2012)

[3] Gu, T., Sun, C., Ma, X., Cao, C., Xu, C., Yao, Y., Zhang, Q., Lu, J., Su, Z.: Practical GUI testing of android applications via model abstraction and refinement. In: Proceedings of the 41st International Conference on Software Engineering, pp. 269–280. IEEE / ACM, Montreal, QC, Canada (2019)

[4] Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z.: Guided, stochastic model-based GUI testing of android apps. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, pp. 245–256. ACM, Paderborn, Germany (2017)

[5] Gu, T., Cao, C., Liu, T., Sun, C., Deng, J., Ma, X., Lu, J.: Aimdroid: Activity-insulated multi-level automated testing for android applications. In: Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution, pp. 103–114. IEEE Computer Society, Shanghai, China (2017)

[6] Espada, A.R., Gallardo, M., Salmerón, A., Merino, P.: Using model checking to generate test cases for android applications. In: Proceedings Tenth Workshop on Model Based Testing. EPTCS, vol. 180, pp. 7–21. London, UK (2015)

[7] Collins, E., Dias-Neto, A.C., Vincenzi, A., Maldonado, J.C.: Deep reinforcement learning based android application GUI testing. In: Proceedings of the SBES '21: 35th Brazilian Symposium on Software Engineering, pp. 186–194. ACM, Joinville, Santa Catarina, Brazil (2021)

[8] Romdhana, A., Merlo, A.: Keynote: ARES: A deep reinforcement learning tool for black-box testing of android apps. In: Proceedings of the 19th IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events, p. 173. IEEE, Kassel, Germany (2021)

[9] Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach. Learn. **8**, 229–256 (1992)

[10] Sutton, R.S., Barto, A.G.: Reinforcement Learning - an Introduction. Adaptive computation and machine learning. MIT Press, ??? (1998)

[11] Degris, T., Pilarski, P.M., Sutton, R.S.: Model-free reinforcement learning with continuous action in practice. In: Proceedings of the 2022 American

Control Conference, pp. 2177–2182. IEEE, Montreal, QC, Canada (2012)

[12] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: Proceedings of the 33nd International Conference on Machine Learning. JMLR.org, New York City, NY, USA (2016)

[13] Köroglu, Y., Sen, A., Muslu, O., Mete, Y., Ulker, C., Tanriverdi, T., Donmez, Y.: QBE: qlearning-based exploration of android applications. In: Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation, pp. 105–115. IEEE Computer Society, Sweden (2018)

[14] Pan, M., Huang, A., Wang, G., Zhang, T., Li, X.: Reinforcement learning based curiosity-driven testing of android applications. In: Proceedings of the ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 153–164. ACM, USA (2020)

[15] Adamo, D., Khan, M.K., Koppula, S., Bryce, R.C.: Reinforcement learning for android GUI testing. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, pp. 2–8. ACM, Lake Buena Vista, FL, USA (2018)

[16] Vuong, T.A.T., Takada, S.: 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, pp. 31–37. ACM, Lake Buena Vista, FL, USA (2018)

[17] Vuong, T.A.T., Takada, S.: Semantic analysis for deep q-network in android GUI testing. In: Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering, pp. 123–170. KSI Research Inc. and Knowledge Systems Institute Graduate School, Hotel Tivoli, Lisbon, Portugal (2019)

[18] EclEmma: JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/index.html

[19] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. CoRR **abs/1606.01540** (2016)

[20] [n.d.]: Appium: Mobile App Automation Made Awesome. http://appium.io/

[21] Developers, A.: Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb

[22] Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: an input generation system for android apps. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 224–234. ACM, Saint Petersburg, Russian Federation (2013)

[23] group, N.: Intent Fuzzer. https://www.nccgroup.trust/us/our-research/intent-fuzzer/

[24] Sasnauskas, R., Regehr, J.: Intent fuzzer: crafting intents of death. In: Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, pp. 1–5. ACM, San Jose, CA, USA (2014)

[25] Ye, H., Cheng, S., Zhang, L., Jiang, F.: Droidfuzzer: Fuzzing the android apps with intent-filter tag. In: Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia, p. 68. ACM, Vienna, Austria (2013)

[26] Hao, S., Liu, B., Nath, S., Halfond, W.G.J., Govindan, R.: PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps. In: Proceedings of 12th Annual International Conference on Mobile Systems, Applications, and Services, pp. 204–217. ACM, Bretton Woods, NH, US (2014)

[27] Mahmood, R., Mirzaei, N., Malek, S.: Evodroid: segmented evolutionary testing of android apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 599–609. ACM, Hong Kong, China (2014)

[28] Anand, S., Naik, M., Harrold, M.J., Yang, H.: Automated concolic testing of smartphone apps. In: Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, p. 59. ACM, Cary, NC, USA (2012)

[29] Mao, K., Harman, M., Jia, Y.: Sapienz: multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 94–105. ACM, Saarbrücken, Germany (2016)

[30] Choi, W., Necula, G.C., Sen, K.: Guided GUI testing of android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, pp. 623–640. ACM, Indianapolis, IN, USA (2013)