Effectively Modeling UI Transition Graphs for Android Apps via Reinforcement Learning

Wunan Guo *[1], Zhen Dong †[1], Liwei Shen †[2], Daihong Zhou ‡[1], Bin Hu §[1], Chen Zhang ¶[1], Hai Xue *[2]
*School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai, China
[1]wnguo@usst.edu.cn, [2]hxue@usst.edu.cn
† School of Computer Science, Fudan University, Shanghai, China
[1]zhendong@fudan.edu.cn, [2]shenliwei@fudan.edu.cn
‡School of Computer Science and Information Engineering, Shanghai Institute of Technology, Shanghai, China
[1]dhzhou@sit.edu.cn
§School of Computer Science, Hangzhou Dianzi University, Hangzhou, China
[1]hubin@hdu.edu.cn
¶School of Computer Science and Technology, Soochow University, Suzhou, China
[1]chenzhang22@suda.edu.cn

2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC) | 979-8-3315-0223-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICPC66645.2025.00011

***Abstract*—Mobile apps are ubiquitous, and have become an indispensable part of our daily life. It is crucial to ensure the correctness, security and performance of these apps through automated GUI modeling. UI Transition Graph (UTG) is an important way of app abstract and modeling. While there have been considerable research efforts on constructing UTG through static or dynamic analysis, obtaining a relatively accurate and complete UTG is challenging. To this end, we present an approach and tool RLDroid that synergistically combines static analysis, dynamic exploration and reinforcement learning techniques to construct UTGs for Android apps. Specifically, RLDroid first extracts a seed UTG through static analysis, and uses this UTG with a depth-first strategy to guide the dynamic exploration. Then, RLDroid provides a Q-learning-based strategy initialized with the generated partial UTG to enhance dynamic exploration and outputs the final UTG. Our experiments on 29 Android apps show that RLDroid identified a total of 871 nodes (i.e., UI pages) and 2726 edges (i.e., transitions) without any false positives, which significantly outperforms the state-of-the-art GUI modeling techniques. Our two exploration strategies, the seed-UTG-guided exploration and the Q-learning-enhanced exploration, make positive contributions to improving the completeness of UTG. Furthermore, the UTGs generated by RLDroid are highly useful for automated GUI testing, resulting in a 60% increase in code coverage and the discovery of 52 additional crashes.**

***Index Terms*—Android apps, Program analysis, GUI testing, Reinforcement learning**

I. Introduction

Mobile applications (apps) are ubiquitous, and have become an indispensable part of our daily life [1], [2]. It is crucial to ensure the correctness, security and performance of these apps through automated GUI modeling, thereby improving their quality and reliability. UI Transition Graph (UTG) is an important way of GUI modeling. In UTG, nodes represent UI pages (e.g., Activity, Fragment, and Menu) and edges represent transitions between UI pages. UTG can be used for functional testing to detect runtime errors [3], [4], for security analysis to identify malicious behavior [5], [6], and for competitive analysis to storyboard app features [7], [8].

While there have been considerable research efforts on constructing UTG [8]–[18] through static or dynamic analysis, obtaining a relatively accurate and complete UTG is challenging. On one hand, due to the inaccuracy of reference analysis and data flow analysis, static analysis generates a large number of false positive transitions [19], and due to a wide range of implementations and code styles, it also misses several transitions [7], [18], [19]. On the other hand, although dynamic analysis can accurately extract transitions, its coverage is far from satisfactory as some states are too deep to explore or require complex inputs [20]. Additionally, dynamic analysis generally takes a long time to obtain the UTG. Some work also employs machine learning [19], [21] to predict transitions between UI pages that cannot be identified by static or dynamic analysis, aiming to enhance the completeness of UTG construction. However, machine learning demands a substantial amount of data, and there can be considerable differences in UI structure among different apps. This leads to the generation of transitions with a higher incidence of false positives and false negatives by such methods.

Reinforcement learning [22]–[24] is a branch of machine learning that does not rely on predefined models or human-made strategies. It focuses on learning optimal strategies through the interaction between agents and the environment to maximize cumulative rewards, and is particularly suitable for solving sequential decision problems. Recently, there have been several studies [24]–[32] utilizing reinforcement learning to dynamically generate UTGs in order to enhance the completeness of UTG construction. However, these methods use random strategies for initialization (i.e., randomly initializing the Q values in Q-table or the parameters of neural networks), which results in slow convergence of the reinforcement learning algorithms and subsequently makes it difficult to adequately explore new UI pages. In addition, for modeling states in reinforcement learning, some methods exhibit overly coarse granularity (e.g., using all widget types on a UI page

2643-7171/25/$31.00 ©2025 IEEE
DOI 10.1109/ICPC66645.2025.00011

to represent a state), which may result in misjudgment of new UI pages. Conversely, other methods exhibit excessively fine granularity (e.g., using the screenshot of a UI page to represent a state), which can generate a substantial amount of redundant states and lead to the state explosion problem. Therefore, it is challenging to find an effective method to build an accurate and complete UTG.

To this end, we propose RLDroid that synergistically combines static analysis, dynamic exploration and reinforcement learning techniques to construct a UTG for an app. Given an app, RLDroid first extracts a seed UTG through static analysis, and uses this UTG to guide the dynamic exploration. In order to obtain a relatively accurate seed UTG, RLDroid does not use traditional pointer analysis techniques, as they generate a large number of false positive transitions [11], [13]. Instead, RLDroid directly analyzes the entry points of event callback methods for widgets and systematically searches the app's global method call graph for API method calls that trigger UI page transitions. These API method calls are provided by Android documentation [33]–[36]. Then, RLDroid parses the parameters in the transition API and combines data flow analysis to obtain the target UI page. For dynamic exploration, RLDroid uses a depth-first strategy to execute as many events as possible to cover more UI pages.

To enhance the completeness of UTG, RLDroid employs Q-learning to augment the dynamic exploration process. Rather than initializing Q-learning with a random strategy, RLDroid utilizes a partially constructed UTG generated from prior static analysis and dynamic exploration for initialization, which can reduce the convergence time of the Q-learning algorithm. Regarding the states in Q-learning, RLDroid models them using a combination of the Activity name, layout tree, and image of the UI page. This method avoids overly coarse state representations while mitigating the problems of state redundancy and explosion caused by excessively fine state representations. Additionally, RLDroid designs a reward function to guide the construction of UTG, which considers the execution frequency of the current event, whether the next state is new, and the execution frequency of all events in the next state. The reward function encourages the Q-learning agent to explore more new UI pages, thereby generating additional transitions. As a result, RLDroid is expected to effectively construct a more accurate and complete UTG.

We evaluate RLDroid on 29 Android apps and compare it with 4 state-of-the-art GUI modeling tools to validate its effectiveness. The results demonstrate that RLDroid surpasses other existing tools in terms of the number of UI pages (871 in total, 158-532 improvement) and transitions (2726 in total, 688-1867 improvement). For the constructed transition edges, RLDroid does not produce any false positives. In addition, we conduct an ablation study to evaluate the contribution of the seed-UTG-guided exploration and the Q-learning-enhanced exploration. The result indicates that the seed-UTG-guided exploration and the Q-learning-enhanced exploration improve the number of identified UI pages by 141% and 101%, respectively, and increase the number of generated transitions by 200% and 149%, respectively. To evaluate the usefulness of RLDroid, we apply it in the automated GUI testing. The result shows that with the guidance of RLDroid, Monkey [37], a widely used GUI testing tool in both academia and industry, achieves a 60% improvement in code coverage and detects 52 additional crashes.

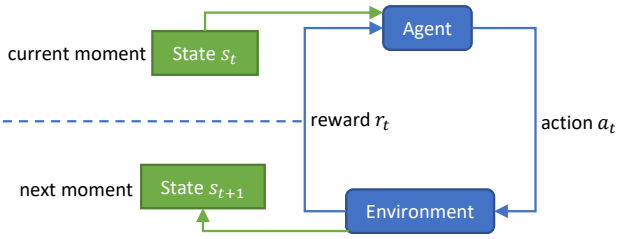

Fig. 1. Markov Decision Process.

Overall, our contributions can be summarized as follows:

- We propose RLDroid, which is a novel approach synergistically combining static analysis, dynamic exploration and reinforcement learning to construct UTGs for Android apps.
- We perform an extensive experiment on 29 Android apps, which demonstrates the effectiveness of RLDroid in UTG construction compared with the state-of-the-art GUI modeling tools. Moreover, experiments show that our reinforcement learning strategy plays a key role in augmenting the completeness of UTG.
- We have implemented RLDroid as an automated tool and make it and the data set used in the experiment are available at https://github.com/RLDroidModeling/RLDroid

II. BACKGROUND

A. Android GUI Modeling

Android Graphical User Interface (GUI) modeling is a crucial aspect of app analysis and security assessments. It involves representing the visual and interactive components of an Android app in a structured format [1], [17]. These components, such as activities, fragments, views, and layout containers, form the backbone of any Android application. The GUI modeling process aims to capture the hierarchy and relationships between these elements, allowing for comprehensive analysis of app behaviors and user interactions [7], [8], [15].

In recent years, Android GUI modeling has evolved to accommodate the dynamic nature of app development. Traditional modeling approaches primarily focused on activities and views, but modern Android apps leverage advanced UI features like fragments and navigation components to enhance user experience. To address these complexities, advanced modeling techniques, including the concept of UI Transition Graphs (UTGs), have emerged. A UTG represents the runtime user interactions within an app as a directed graph, capturing the transitions between different UI pages. This model provides a comprehensive understanding of app flow and behavior, enabling more precise analysis and detection of non-compliant or malicious behaviors [5], [38].

B. Q-learning

Q-learning [22] is a popular reinforcement learning algorithm that is model-free and off-policy. It aims to learn the optimal policy that maximizes the expected cumulative reward through interaction with the environment. Q-learning operates by maintaining a Q-table, which maps state-action pairs to their estimated Q-values, representing the expected utility of taking an action in a particular state.

Reinforcement learning problems can be represented as a standard Markov Decision Process (MDP), as shown in Figure 1. At each step, the agent observes the current state s_t and selects an action a_t based on the Q-values. The environment then transitions to a new state s_{t+1} and provides an immediate reward r_t. The Q-values are updated according to the Bellman equation, which captures the relationship between the current Q-value and the optimal future Q-value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right] \quad (1)$$

where s_t and a_t are the state and action at time t respectively, r_t is the immediate reward received after taking action a_t in state s_t, α is the learning rate from 0 to 1, γ is the discount factor that determines the importance of future rewards which usually between 0.8 and 0.99 [27], [29], and a' represents all possible actions in the next state s_{t+1}. Over time, as the agent continues to interact with the environment and update its Q-values, the Q-table converges to represent the optimal policy [39]. Therefore, if the agent selects the action with the highest Q-value in each state, the goal of maximizing cumulative reward can be achieved.

III. Motivating Example

AnkiDroid[1] is a highly popular flashcard app with 8,400 stars on Github and over 10 million downloads on Google Play. RLDroid constructs a UTG for this app, where 57 nodes are used to represent the UI pages and 383 edges are used to represent the transitions among them. Due to space limitation, Figure 2 shows part of UTG of the app, which includes 6 UI pages (p1-p6) and 6 transitions (solid black edges). The label on each edge indicates the event or widget that triggers this transition. For instance, after clicking the menu button on the top right of page p1, the menu will open, and the page will navigate from p1 to p2. When the Back button on the device is pressed, the page will return to p1.

There are difficulties in building a relatively accurate and complete UTG. For AnkiDroid, we first leverage the state-of-the-art static analysis tool, GoalExplorer [15], to construct its UTG, resulting in 23 nodes and 492 transition edges. Unfortunately, among these 492 edges, 419 are false positives. The reason for so many false positives lies in the deficiency of reference analysis. GoalExplorer employs overly approximate data flow analysis to calculate potential transitions between two UI pages. In Figure 2, the dashed red line represents a false positive transition generated by GoalExplorer. This transition is triggered by clicking on the list item "Advanced

[1]AnkiDroid. https://play.google.com/store/apps/details?id=com.ichi2.anki

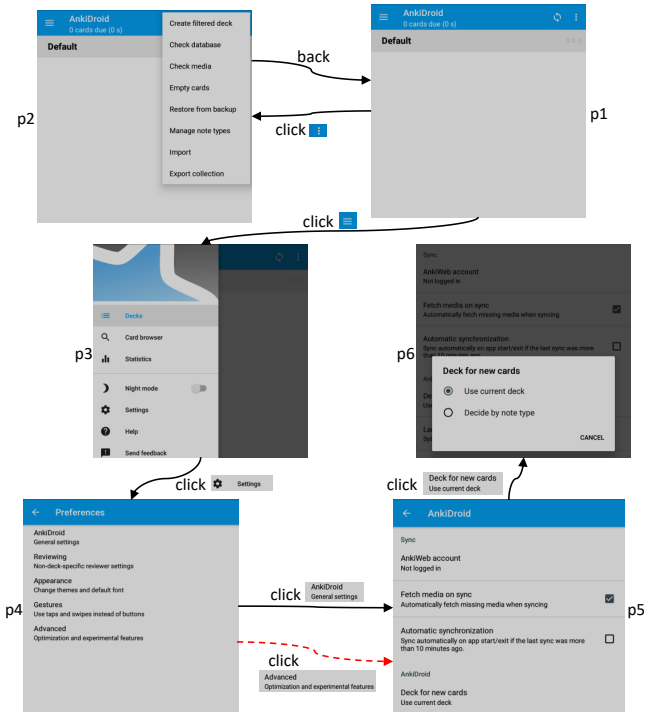

Fig. 2. Partial UTG of AnkiDroid.

Optimization and experimental features", navigating from page p4 to p5, while the correct transition should be triggered by clicking on the list item "AnkiDroid General setting". False positive transitions not only mislead the navigation between their source and target pages but also affect subsequent transitions starting from the target page. Additionally, GoalExplorer only identifies 23 UI pages (i.e., 23 nodes) because it is unable to recognize dynamically rendered UI pages. To reduce the false positive transitions, some tools employ dynamic or hybrid analysis to construct UTG, such as Promal [19], SceneDroid [18], and Q-testing [27]. We use these three tools to construct the UTG for AnkiDroid, but none of them are able to recognize the transition between p4 and p5 in Figure2. On the other hand, Q-testing, Promal, and SceneDroid can only identify 13, 19, and 27 UI pages respectively, along with 87, 104, and 235 transitions, due to the weakness of low page coverage in dynamic analysis. Additionally, none of these three tools can effectively identify duplicate UI pages, which results in a large number of redundant nodes in the constructed UTG.

To enhance the accuracy and completeness of UTG while minimizing redundant nodes, RLDroid constructs it through the following steps:

- ***Seed UTG Extraction.*** RLDroid utilizes the app's global transition graph and APIs related to page transitions in the Android documentation (e.g., Activity transitions and opening dialogs) to construct a relatively accurate seed UTG.
- ***Seed-UTG-Guided Exploration.*** RLDroid employs the

seed UTG to guide dynamic exploration and combines a depth-first strategy to execute as many events as possible. Through the dynamic exploration guided by the seed UTG, a partial UTG is established.

- ***Q-learning-Enhanced Exploration.*** RLDroid initializes the Q-learning algorithm with the partial UTG, thereby perpetuating the dynamic exploration process. Each UI page is represented by its Activity name, layout tree, and image, serving as the foundation for state modeling in Q-learning and facilitating the identification of identical pages. Furthermore, RLDroid introduces an exploration reward function aimed at incentivizing the discovery of a greater number of UI pages.

IV. Approach

Figure 3 describes the workflow of RLDroid. It comprises two steps: (1) static analysis and (2) dynamic exploration. Given an APK, RLDroid first uses static analysis to extract UI pages of the app, identify transitions between these pages, and construct a seed UTG. Guided by this seed UTG, RLDroid subsequently employs a dynamic exploration method that integrates depth-first search and Q-learning-based strategies to thoroughly navigate the app. This process results in the output of a comprehensive UTG, accompanied by detailed analytical findings, such as the screenshot and layout tree corresponding to each individual UI page.

A UTG is defined as $G = (U, E, \Sigma)$, where:

- Node $u \in U$ indicates a UI page, and we consider 4 categories of UI pages that users can interact with as a node in a UTG: activities, fragments, menus, and dialogs. Activities and fragments are typically rendered in the form of full-screen pages. Menus (e.g., Options Menu and Context Menu) and dialogs are short-lived windows that often require the user to take actions.
- Edge $e \in E$ is an edge between nodes representing a page transition.
- Label $\sigma \in \Sigma$ is a label on an edge representing a event that triggers the page transition. σ is a tuple $< w, type >$ where w is the widget that triggers σ and $type$ is the type of this event σ, such as $click$ and $long\_click$.

A. Static Analysis

Static analysis aims to analyze the UI pages, widgets and inter-page transitions within an app, thereby facilitating the construction of a seed UTG. This process encompasses two pivotal steps: UI extraction and transition identification.

UI Extraction. Based on our definition of UTG, 4 types of UI pages are considered: `Activity`, `Fragment`, `Menu` and `Dialog`. We extract all activities from the *AndroidManifest.xml* file and search for the corresponding activity classes by their names. For the identification of fragments, we filter out classes that directly or indirectly extend `android.app.Fragment` from the app code. Similarly, we extract dialogs by searching for classes that inherit from `android.app.Dialog`. Menus are hosted in an activity and are initialized by callback methods such as `onCreateOptionsMenu` and `onCreateContextMenu`. Therefore, we check whether there are such methods in the code of each activity in order to extract the menu-type page.

TABLE I
API Methods related to activity, fragment, menu, and dialog transition

Transition Type	Class	Method
Activity	android.app.Activity	startActivity*
Fragment	*.FragmentTransation	add attach
Menu	android.app.Activity *.Fragment	onCreateOptionsMenu
	android.app.Activity *.Fragment	onCreateContextMenu
	android.app.Activity *.Fragment	registerForContextMenu
Dialog	android.app.Dialog	show

Apart from extracting UI pages, we also retrieve all the widgets within each page. For the widgets in activities, fragments or dialogs, they can be defined within the corresponding XML layout file or dynamically added to the UI page through tailored code statements. To obtain these widgets and their event handlers, we firstly identify events associated with widgets registered in the layout file, for example, by retrieving configurations like `android:onClick="onClick"`. This allows us to obtain the widget's resource ID, type, and text, as well as locate the callback method for the event. Secondly, we recognize events that are specified using listener registration methods, such as `setOnClickListener`, by scanning the app code. Furthermore, we apply data-flow analysis to the method's caller to pinpoint its declaration statement. This could be a `findViewById` invocation if the widget is specified in the layout, or a `new` instantiation method if the widget is created dynamically. In such cases, we obtain the widget's resource id (if provided) and retrieve its type and text either from the layout file or from the arguments of the method invocations.

Menus are typically structured as hierarchical arrangements of menu items, with each item considered a widget equipped with a corresponding event handler. The interface of a menu can be defined through both static and dynamic means. Static definition involves specifying the components of a menu within a resource file and subsequently loading this resource into the program. Conversely, dynamic definition employs menu-specific methods, such as `add` and `addSubMenu`, to construct the menu's architecture. In contrast to other types of widgets, deep-level menu items in a multi-level menu are only visible when their parent menus are selected. Therefore, when identifying hierarchical menu items, we also record the display path for each submenu item.

Transition Identification. The transitions between different UI pages are generally identified by some pre-specified API methods invoked in the callback of a widget. These methods are related to UI transitions and Table I shows the commonly used API methods related to `Activity`, `Fragment`, `Menu`, and `Dialog` transition. Due to limited space, we use asterisks

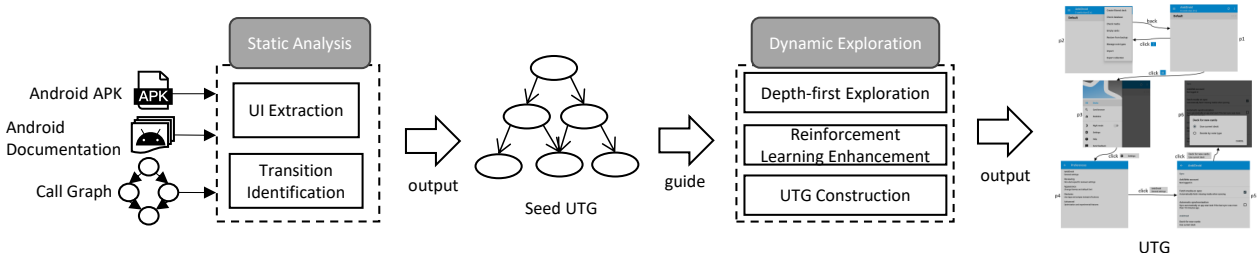

Fig. 3. The workflow of RLDroid.

to omit some prefix and suffix letters. We search within the callback methods of the widgets to see if there are any invocations of such API methods. Additionally, considering that such transition-related APIs may not be directly called within the callback methods (i.e., they may be called through one or more layers of other custom methods), we leverage the app's method call graph to facilitate our search and matching process. Specifically, we initially acquire the app's comprehensive method call graph using FlowDroid [40]. Subsequently, within this graph, we systematically search for paths originating from callback methods (e.g., `onClick`) and terminating at transition-related API methods (e.g., `startActivity`). Upon successful identification of such paths, we employ data flow analysis to ascertain the calling objects and parameters of the transition API, thereby extracting the target page. We then establish a transition edge predicated on the source page, target page, the corresponding widget, and event information. Once all UI pages have been analyzed, we integrate all identified transitions to construct the app's seed UTG. Please note that the constructed seed UTG is incomplete due to limitations of static analysis. We only use it to guide dynamic exploration to generate a more complete UTG.

B. Dynamic Exploration

The goal of dynamic exploration is to explore more UI pages and uncover additional transitions by using the guidance of the seed UTG and enhancements from Q-learning, thereby generating a more complete UTG. We employ the Markov Decision Process (MDP) to formulate the dynamic exploration. Therefore, in this section, we first introduce the definitions of each element in the MDP quadruplet $< S, A, P, R >$, and then describe the process of dynamic exploration in detail.

State Space: S**.** Many studies on automated testing have proposed methods for representing a state of a UI page [9], [14], [20], [41], [42]. These methods encompass utilizing the layout, the image, and the widget information (e.g., widget types) of a UI page to represent a state. However, widget information provides a too coarse representation of states, as many different pages may share the same widget information. Although page layouts or page images can represent states in a fine-grained manner, minor changes in the content of a page can lead to changes in page layout and page image, resulting in state redundancy and explosion problems. To this end, we represent a state by integrating the layout with the image of the page, achieving a fine-grained representation while significantly mitigating the problems of state redundancy and explosion. More specifically, for a given state s_t within S, it is characterized as a triplet $< a_t, l_t, im_t >$. Here, a_t denotes the name of the activity associated with s_t. The l_t component signifies the structural arrangement of s_t, manifested as a widget tree. Within this hierarchical structure, leaf nodes embody non-container widgets (e.g., buttons), whereas non-leaf nodes represent container widgets (e.g., linear layouts). In the process of state modeling, certain widget attributes, such as text, are deliberately omitted. This omission serves dual purposes: simplifying the layout and mitigating the generation of excessive redundant states. Lastly, im_t represents the visual snapshot of s_t at time t, captured as an image of the page. For two states s_i and s_j, their similarity $sim(s_i, s_j)$ is defined as equation 2.

$$sim(s_i, s_j) = \begin{cases} \alpha sim(l_i, l_j) + \beta sim(im_i, im_j) & a_i = a_j \\ 0 & a_i \neq a_j \end{cases} \quad (2)$$

When the respective activity names associated with s_i and s_j differ, the similarity between them is 0. Conversely, the similarity is computed as the weighted average of their layout similarity and image similarity. In equation 2, $sim(l_i, l_j)$ denotes the layout similarity between s_i and s_j, while $sim(im_i, im_j)$ represents the image similarity. The weighting coefficients are designated as α and β ($\alpha + \beta = 1$), respectively. In our practice, we use the normalized tree edit distance [43] to calculate the layout similarity and use the image matching algorithm in OpenCV [44] to calculate the image similarity. With regard to the weighting coefficients, we set the values of α and β to 0.7 and 0.3, respectively.

Action Space: A**.** User interaction events within the app are formulated as actions within the MDP action space. For simplicity, no distinction is made between events and actions. Events in the action space A originate from two parts: one is the events obtained through static analysis in the seed UTG, and the other is the events inferred from the properties of widgets on the page (e.g., clickable, scrollable). Since each event is associated with a specific state, we use a state-event pair (s, a) to represent an executable event, and the value of the event (i.e., Q-value) is stored in the Q-table.

State Transition Function: P**.** The implementation logic of the app can be regarded as the transition function P, which determines the subsequent state after executing action a_t.

Reward Function: R**.** The role of the reward function is to generate a numerical value to guide the agent in conducting valuable explorations and triggering more new transitions. It is defined by equation 3 and consists of three parts. Where,

- $|s_t, a_t|$ denotes the execution frequency of a_t in s_t, and a lower execution frequency implies a higher value attributed to the action a_t.
- $|s_{t+1}|$ represents the number of unexecuted events in s_{t+1}. The more unexecuted events, the higher the value of action a_t. It should be noted that for a certain event a_{t+1} in s_{t+1}, if it triggers the transition to an activity or fragment, its execution flag is set to "executed" immediately after its execution. However, if it triggers the display of a menu or dialog, a_{t+1} is only marked as "executed" after all events in the menu or dialog have been fully executed.
- $isNew(s_{t+1})$ denotes whether s_{t+1} is a new state. If it is, $isNew(s_{t+1})$ takes the value of 1; otherwise, it is 0. We record the visited states during the exploration and calculate the similarity between s_{t+1} and these states according to equation 2. If the similarity between s_{t+1} and all previous states is below a given threshold (which we set as 0.99 based on experience), then s_{t+1} is considered a new state.

$$R = \frac{1}{|s_t, a_t|} + |s_{t+1}| + isNew(s_{t+1}) \tag{3}$$

Dynamic Exploration Process. Algorithm 1 depicts the detailed dynamic exploration process, which includes two phases: exploration guided by seed UTG (lines 3-19) and exploration enhanced by Q-learning (lines 21-29). The exploration guided by seed UTG adopts a depth-first strategy. For each interaction, RLDroid first extracts the current state s_t (line 4). If s_t is a new one, RLDroid adds it to the state set U (line 5). Then, RLDroid searches for the corresponding page node in the seed UTG based on the activity name and layout of s_t, and selects an unexecuted event σ from it (line 6). If σ does not exist in s_t or all events in the seed UTG have been executed, RLDroid selects an unexecuted event from s_t in a top-down search manner. If all events in s_t have been executed or no events can be selected, it executes a back event to return to the previous state (lines 7-14). After executing σ, RLDroid calculates the reward for this event according to equation 3 (if it is a back event, the reward is 0), updates the Q-table according to equation 1, and updates the edges and events in the UTG (lines 15-18).

$$chooseAction(s) = \begin{cases} \underset{a}{argmax}\ Q(s,a) & 1-\epsilon \\ random\ action & \epsilon \end{cases} \tag{4}$$

The exploration process enhanced by Q-learning is similar to that guided by the seed UTG. For each interaction, RLDroid first obtains the current state (line 22), updates the state set (line 23), selects and executes an action (lines 24-25), then calculates the reward for the action (line 26), and finally updates the Q-table as well as the edges and events in the UTG (lines 27-28). The only difference lies in the action selection strategy employed by RLDroid, which uses the ϵ-greedy strategy defined in equation 4: it selects the action with the highest Q-value in s_t with a probability of $1-\epsilon$ and selects a random action with a probability of ϵ. Based on previous experience [27], [31], we set ϵ to 0.2. This strategy guides RLDroid to explore in the direction of increasing rewards while encouraging it to explore unknown states. It should be noted that if an app crash is encountered during exploration, RLDroid will restart the app and continue the exploration.

Algorithm 1: Dynamic Exploration

Input: Seed UTG: G, $timeout$
Output: UTG

1 $U, E, \Sigma, Q \leftarrow \{\}$;
2 launch app ;
3 **repeat**
4 $s_t \leftarrow$ getCurrentState();
5 $U \leftarrow U \cup \{s_t\}$;
6 $\sigma \leftarrow$ selectEvent(G, s_t) ;
7 **if** $\neg\sigma.exist()$ **then**
8 **if** *allEventsExecuted*(s_t) **then**
9 $\sigma \leftarrow back$;
10 **end**
11 **else**
12 $\sigma \leftarrow$ selectUnexecutedEvent(s_t) ;
13 **end**
14 **end**
15 $s_{t+1} \leftarrow$ execute(σ) ;
16 $r_t \leftarrow$ getReward(σ, s_{t+1}) ;
17 $Q(s_t,\sigma) \leftarrow Q(s_t,\sigma) + \alpha\left[r_t + \gamma \max_{\sigma'} Q(s_{t+1},\sigma') - Q(s_t,\sigma)\right]$;
18 $e \leftarrow < s_t, s_{t+1}, \sigma >$, $E \leftarrow E \cup \{e\}$, $\Sigma \leftarrow \Sigma \cup \{\sigma\}$;
19 **until** $timeout$;
20 restart app ;
21 **repeat**
22 $s_t \leftarrow$ getCurrentState();
23 $U \leftarrow U \cup \{s_t\}$;
24 $a_t \leftarrow$ chooseAction(s_t) ;
25 $s_{t+1} \leftarrow$ execute(a_t) ;
26 $r_t \leftarrow$ getReward(a_t, s_{t+1}) ;
27 $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right]$;
28 $e \leftarrow < s_t, s_{t+1}, a_t >$, $E \leftarrow E \cup \{e\}$, $\Sigma \leftarrow \Sigma \cup \{a_t\}$;
29 **until** $timeout$;
30 $UTG \leftarrow < U, E, \Sigma >$;

V. Implementation

RLDroid is implemented as a fully automated UTG modeling framework, which reuses or extends a set of off-the-shelf tools: Soot [45], FlowDroid [40], UI Automator [46], Android Debug Bridge (ADB) [47], and Logcat [48]. Soot is used to analyze the intermediate code of an app and extract its XML resource files. FlowDroid is extended to build the call graph of an app and perform data flow analysis. UI Automator is used to dump GUI hierarchy and screenshot of UI pages and perform dynamic exploration. ADB is used to send events and obtain the activity name of the current UI page. During exploration, we use Logcat to monitor and record runtime exceptions.

VI. Evaluation

In our experimental evaluation, we seek to answer the following three research questions:

- **RQ1:** How effective is RLDroid in constructing UTGs, compared to existing GUI modeling tools?
- **RQ2:** How much do the seed-UTG-guided exploration and the Q-learning-enhanced exploration contribute to UTG construction?
- **RQ3:** How useful is RLDroid in automated GUI testing in Android apps?

A. Evaluation Setup

Subject Apps. We evaluated RLDroid on a data set containing 29 apps which were collected by the following three steps. Firstly, we collected 78 non-duplicated apps from two Android testing benchmarks, ANDROTEST [3] and THEMIS [49], which have been used in several studies [20], [24], [27], [42]. Secondly, we excluded 29 apps that have not been maintained in the last two years in order to receive timely feedback from app developers, as well as 23 apps with fewer than 2 activities. Finally, we added 3 apps downloaded from Google Play, which were used in the early research investigations of our project (marked by "#" in Table II), to the data set. Similar to Table I, we use asterisks to omit some suffix letters of app names due to space limitations.

Execution Environment. Our experiments ran on a 64-bit Ubuntu physical machine with 2.50GHz Intel(R) Core(TM) i9-12900H CPU and 32GB RAM, and used an Android emulator to perform dynamic exploration. The emulator was configured with 2GB RAM and Android Nougat operating system (SDK 7.1, API level 25).

Evaluation Setup of RQ1. To evaluate the effectiveness of RLDroid in constructing UTG, we compare it with 4 representative GUI modeling tools: GoalExplorer [15], Q-testing [27], PROMAL [19], and SceneDroid [18]. GoalExplorer is the most recent technique that utilizes static analysis to construct the UTG of an app. Q-testing dynamically constructs the UTG using reinforcement learning, and it designs a curiosity-driven exploration strategy to select events that are most likely to navigate to new pages. PROMAL and SceneDroid are two recent techniques that combine static and dynamic analysis to construct a UTG. PROMAL uses static analysis to construct a static UTG, then employs dynamic analysis to verify the transitions within it, and leverages machine learning to predict feasible transitions. SceneDroid, on the other hand, uses static analysis to extract ICC messages and construct an Activity Transition Graph (ATG). It then combines dynamic analysis with indirect activity launches to generate a more comprehensive UTG.

For each transition in the UTG, we classify it into two categories:

- *True Positive (TP):* After executing the event on the transition, the app should navigate from the source page of the transition to the target page.
- *False Positive (FP):* After executing the event on the transition, the app should not navigate from the source page of the transition to the target page.

We manually check whether a transition is a false positive. First, we navigate the app to the source page based on information such as the page's activity name and screenshot. Then, we identify the widget based on its information (e.g., widget id and widget image) and execute the corresponding event. Finally, we check whether the app jumps to the target page. If it jumps to the target page, we consider the transition as a true positive; otherwise, it is a false positive.

We allocated 1 hour for each evaluated tool in one run, and repeated 5 independent runs for each tool. Regarding RLDroid, we allocated 30 minutes for the seed-UTG-guided exploration and 30 minutes for the Q-learning-enhanced exploration.

Evaluation Setup of RQ2. To evaluate the contribution of different strategies in UTG construction, we implemented two versions of RLDroid (RLDroid$^{\alpha}$ and RLDroid$^{\beta}$) as the baselines and conducted an ablation study.

- RLDroid$^{\alpha}$: This baseline comparison aims to evaluate the contribution of the exploration guided by seed UTG. Specifically, RLDroid$^{\alpha}$ removes the seed-UTG-guided exploration in Algorithm 1 (lines 3-19), and only uses the Q-learning-enhanced strategy to interact with the app. For each interaction, RLDroid$^{\alpha}$ determines the optimal event on the UI page by utilizing the ϵ-greedy strategy as outlined in equation 4.
- RLDroid$^{\beta}$: This baseline comparison aims to evaluate the contribution of the exploration enhanced by Q-learning. Specifically, RLDroid$^{\beta}$ removes the Q-learning-enhanced exploration in Algorithm 1 (lines 21-29) and solely employs the seed-UTG-guided strategy to interact with the app. For each interaction, RLDroid$^{\beta}$ adopts a depth-first strategy guided by the seed UTG (lines 6-14 of Algorithm 1) to select the optimal event on the UI page.

We allocated 1 hour for RLDroid$^{\alpha}$ and RLDroid$^{\beta}$. If RLDroid is capable of constructing a more complete UTG than RLDroid$^{\alpha}$ (or RLDroid$^{\beta}$), we can conclude that the exploration guided by a seed UTG (or enhanced by Q-learning) plays an effective role in the construction of UTG.

Evaluation Setup of RQ3. In order to further evaluate the usefulness of RLDroid, we applied it to guide automated GUI testing. We used Monkey [37], which is a widely-used lightweight Android test input generator in both academia and industry [3], [4], as an example to modify it for experiments. Specifically, for each app, we tested it using the Monkey with and without the UTG generated by RLDroid for 1 hour respectively. We evaluate the usefulness of RLDroid based on the coverage and the number of detected crashes. If Monkey with the guidance of RLDroid achieves higher coverage and discovers more crashes compared to Monkey itself, we can conclude that RLDroid is indeed useful for automated GUI testing.

B. Experimental Results

RQ1: Effectiveness. Table II shows the results of RLDroid and 4 state-of-the-art GUI modeling tools (GoalExplorer, PROMAL, Q-testing, and SceneDroid) on 29 Android apps. Column "#N" indicates the total number of nodes in the UTG built by each tool. Column "#E" indicates the total number of edges identified by each tool. Column "TP" and

TABLE II
RESULTS OF RLDROID (RD), GOALEXPLORER (GE), PROMAL (PM), Q-TESTING (QT), SCENEDROID (SD), RLDROID$^{\alpha}$ (RD$^{\alpha}$), AND RLDROID$^{\beta}$ (RD$^{\beta}$).

App	RD					GE				PM				QT		SD		RD$^{\alpha}$		RD$^{\beta}$	
	#N	#E	TP	FP	SA(s)	#N	#E	TP	FP	#N	#E	TP	FP	#N	#E	#N	#E	#N	#E	#N	#E
ActivityDiary	20	73	73	0	12	9	64	11	53	11	58	49	9	9	28	17	55	11	32	14	40
AmazeFile*	25	47	47	0	6	8	52	13	39	15	49	31	18	12	16	23	41	13	19	15	24
AnkiDroid	57	383	383	0	49	23	492	73	419	19	122	104	18	13	87	27	235	21	99	25	107
Collect	36	51	51	0	10	19	82	21	61	28	53	37	16	20	29	31	43	22	32	26	39
FirefoxLite	15	26	26	0	4	6	30	5	25	10	27	18	9	5	4	15	23	8	11	8	11
Geohash*	19	36	36	0	5	8	17	12	5	13	30	28	2	5	9	16	29	7	13	8	16
Material*	13	27	27	0	11	5	15	7	8	7	20	16	4	6	8	11	19	6	9	7	10
OmniNotes	28	55	55	0	16	10	63	17	46	20	61	48	13	7	10	24	50	8	10	12	16
Osmeditor	75	286	286	0	39	28	342	85	257	53	196	168	28	22	51	63	201	20	47	25	56
Phonograph	18	31	31	0	7	7	40	11	29	16	27	27	0	6	8	18	29	6	7	9	11
ScarletNotes	31	78	78	0	9	16	105	34	71	25	70	62	8	14	30	30	75	18	39	18	42
#BeeCount	14	22	22	0	4	4	11	4	7	9	26	19	7	5	5	12	20	5	6	8	9
#Glucosio	29	53	53	0	13	10	67	19	48	23	47	44	3	9	15	24	47	12	23	12	25
#Gnucash	33	77	77	0	35	24	96	31	65	26	38	34	4	18	26	30	69	16	25	16	27
WorldClock	13	23	23	0	6	5	20	7	13	8	28	19	9	4	5	10	21	6	9	6	11
AntennaPod	57	193	193	0	40	26	254	58	196	41	148	127	21	23	54	46	172	20	50	25	58
36C3*	12	18	18	0	30	4	11	4	7	9	21	13	8	3	5	12	16	4	6	6	7
AnyMemo	40	78	78	0	34	23	69	45	24	32	75	66	9	18	37	35	72	21	40	24	43
Aphoto*	37	79	79	0	23	21	73	48	25	30	74	64	10	20	46	33	70	17	39	24	48
ArxivMobile	13	20	20	0	8	5	23	7	16	10	26	17	9	6	9	11	15	7	9	8	11
BookCata*	43	218	218	0	37	17	311	89	222	29	206	188	18	21	102	36	179	20	98	25	124
Dalvik*	13	18	18	0	5	3	16	4	12	10	22	14	8	3	3	12	16	4	4	6	7
Feeder	11	17	17	0	8	5	26	7	19	7	23	15	8	4	5	9	15	5	5	5	6
K9	48	136	136	0	47	19	286	82	204	37	158	120	38	24	89	40	125	20	87	22	93
Keepass*	9	14	14	0	16	5	24	7	17	8	17	12	5	3	5	9	13	6	8	7	9
MyExpenses	81	416	416	0	58	27	591	106	485	63	242	199	43	30	121	60	187	26	120	30	133
RingDroid	9	14	14	0	8	4	19	4	15	7	15	12	3	5	6	7	12	5	5	6	6
RunnerUp	52	198	198	0	36	17	192	38	154	34	170	149	21	15	33	41	172	18	42	27	85
Wikipedia	20	39	39	0	52	6	45	10	35	13	32	24	8	9	16	11	17	9	16	10	21
Sum/Avg	871	2726	2726	0	22	364	3436	859	2577	613	2081	1724	357	339	862	713	2038	361	910	434	1095

"FP" categorizes whether the edges are true positives or false positives. It is worth noting that Table II does not show TP and FP for Q-testing and SceneDroid, as both tools dynamically construct UTGs, where all edges are true positives. Therefore, there is no necessity for TP and FP columns in this context. Column "SA(s)" indicates the time spent by RLDroid to construct the seed UTG through static analysis.

RLDroid generated a total of 871 nodes and 2726 edges in the 29 Android apps, with no false positives among these edges. In comparison with the state-of-the-art techniques, RLDroid identifies the largest number of nodes and edges, followed by SceneDroid (713 nodes and 2038 edges), PROMAL (613 nodes and 1724 edges), GoalExplorer (364 nodes and 859 edges), and Q-testing (339 nodes and 862 edges). Additionally, RLDroid constructs the UTG with the highest number of nodes and edges on all 29 apps. Regarding false positives, GoalExplorer identified 3436 edges, but 2577 are false positives. GoalExplorer yielded a significant number of false positive edges because it uses static analysis to infer possible transitions and is affected by the low precision problem in static analysis. PROMAL also yielded several false positive edges, and 357 out of 2081 identified edges are false positives. PROMAL uses machine learning to predict transitions between UI pages, and due to the limitations of the dataset and machine learning algorithms, this can also result in some false positives. RLDroid does not generate false positive transitions because it uses dynamic exploration to generate the final UTG.

For SceneDroid, although it enhances dynamic exploration by leveraging ICC information and strategies for indirectly launching activities, it still faces the problem of low UI page coverage. Moreover, indirectly launching apps from non-main activities sometimes fails. Q-testing utilizes Q-learning to enhance dynamic exploration, but its primary goal is GUI testing to discover crashes, rather than constructing a complete UTG. Additionally, Q-testing initializes its Q-table using a random strategy, leading to slow convergence. Consequently, it fails to identify many UI pages and transitions, and the number of nodes and edges identified by it is only about 39% and 32% of RLDroid, respectively. RLDroid uses the seed UTG obtained through static analysis to guide dynamic exploration and explores more UI pages in combination with a depth-first strategy. Meanwhile, RLDroid uses the partially constructed UTG to initialize the Q-table to accelerate its convergence. Additionally, RLDroid designs a reward function to drive it to discover more unvisited UI pages and transitions. Therefore, compared to other tools, the UTG constructed by RLDroid is more complete, containing more nodes and edges.

> RLDroid yielded a total of 871 nodes and 2726 edges in the 29 apps, with no false positive edges. RLDroid significantly outperforms the state-of-the-art GUI modeling techniques in terms of the UTG's accuracy and completeness.

RQ2: Contribution of Exploration Strategies. The last four colunms in Table II shows the number of nodes and edges in the UTG generated by RLDroid$^{\alpha}$ and RLDroid$^{\beta}$. RLDroid$^{\alpha}$ generated a total of 361 nodes and 910 edges, whereas RLDroid generated approximately 2.4 times and 3 times the number of nodes and edges, respectively. Additionally, for each app, RLDroid identifies a significantly higher number of

nodes and edges compared to RLDroid$^{\alpha}$. Therefore, we can see that the exploration strategy guided by seed UTGs plays a crucial role in the construction process of UTGs. RLDroid$^{\beta}$ generated a total of 434 nodes and 1095 edges, while RLDroid produced approximately twice and 2.5 times the number of nodes and edges, respectively. Meanwhile, for each app, the number of nodes and edges identified by RLDroid is also significantly greater than that of RLDroid$^{\beta}$. Thus, the Q-learning-enhanced exploration also plays an effective role in the construction of UTGs.

For the exploration guided by seed UTGs, RLDroid leverages the seed UTGs to quickly discover new UI pages and can execute events on these new UI pages to access adjacent pages. In this way, RLDroid continuously iterates its exploration, ultimately significantly increasing the number of identified UI pages and transitions. On the other hand, when RLDroid gets stuck in some repeated UI pages, the depth-first strategy can help it quickly jump out of the current page to conduct new explorations, which also improves the efficiency of discovering new pages. Regarding the overhead of statically constructing seed UTGs, RLDroid takes an average of 22 seconds, which is considered acceptable.

For the Q-learning-enhanced exploration, RLDroid utilizes the partially constructed UTG during the exploration process to initialize the Q-table, which makes the Q-value of each event in the Q-table relatively accurate and thus accelerates the discovery of new UI pages. Additionally, the reward function we designed also guides RLDroid to explore towards new UI pages. On the other hand, the total number of nodes and edges generated by RLDroid$^{\alpha}$ is 22 and 48 more than those generated by Q-testing, respectively, which also indicates that the Q-learning-enhanced exploration strategy of RLDroid has been improved compared to traditional Q-learning based exploration.

Both the seed-UTG-guided and Q-learning-enhanced explorations play effective roles in the construction of UTG, and they both improve the completeness of UTG.

RQ3: Usefulness. Table III shows the coverage (%Cov) and the number of crashes (#Cra) for each app tested with Monkey guided by RLDroid (MO with RD), and Monkey without RLDroid (MO without RD). The Monkey guided by RLDroid achieved an average coverage of 50.3%, which is 1.6 times that of the Monkey without RLDroid, and it only achieved an average coverage of 31.4%. Across all apps evaluated, the coverage of the Monkey guided by RLDroid consistently surpassed that of the unguided Monkey, with the magnitude of this improvement varying between 5 and 43 percentage points. This demonstrates the efficacy of the more comprehensive UTG produced by RLDroid in guiding automated testing, i.e., providing a comprehensive perspective for exploring the app during automated testing.

With RLDroid, Monkey detected a total of 67 crashes from 29 apps, which is 4.5 times more than that without RLDroid. The Monkey without RLDroid only found 15 crashes, and for 20 apps, it failed to detect any crashes. For each app, in terms of crash detection, Monkey guided by RLDroid performs better (at least not worse) than Monkey without RLDroid. Therefore, the UTGs constructed by RLDroid can help automated GUI testing tools find more bugs.

TABLE III
RESULTS OF GUIDING AUTOMATED GUI TESTING TOOL

Id	App	MO with RD		MO without RD	
		%Cov	#Cra	%Cov	#Cra
1	ActivityDiary	40	2	7	0
2	AmazeFileManager	30	2	21	1
3	AnkiDroid	43	4	24	2
4	Collect	64	0	45	0
5	FirefoxLite	50	7	36	2
6	Geohashdroid	36	2	11	1
7	MaterialFBook	38	0	29	0
8	OmniNotes	53	3	41	0
9	Osmeditor	45	6	21	2
10	Phonograph	39	0	33	0
11	ScarletNotes	55	4	28	0
12	BeeCount	71	1	53	0
13	Glucosio	59	2	18	0
14	Gnucash	60	1	37	0
15	WorldClock	98	1	92	0
16	AntennaPod	59	4	30	1
17	36C3 Schedule	55	0	47	0
18	AnyMemo	40	0	30	0
19	AphotoManager	61	7	37	2
20	ArxivMobile	53	0	40	0
21	BookCatalogue	46	4	35	3
22	DalvikExplorer	70	0	52	0
23	Feeder	39	0	12	0
24	K9	16	8	6	0
25	Keepassdroid	31	1	7	0
26	MyExpenses	60	2	49	1
27	RingDroid	63	2	20	0
28	RunnerUp	47	4	19	0
29	Wikipedia	37	0	32	0
Avg/Sum		50.3	67	31.4	15

We give an example from our evaluated apps to describe the usefulness brought by RLDroid. BookCatalogue [2] is a popular book cataloguing app with over 500 thousand downloads on Google Play. We trigger a detected crash by the following steps. First, click the "Add Book" button on the homepage, which will bring up a dialog for adding a book. Subsequently, click the "Add Manually" button in the dialog, and the book details page will appear. Next, select and click the "NOTES" tab on this page, which will switch to a note fragment. Upon interacting with the rating bar within this Fragment, we then click on the menu button positioned at the top right corner of the screen and select the "Share" menu option. At this point, the app crashes. Monkey failed to uncover this bug, as it got stuck on the homepage. For RLDroid, it constructed a UTG with 43 nodes and 218 edges. Guided by this UTG, Monkey was able to successfully execute the aforementioned steps, navigating to the note fragment, selecting the "Share" menu item, and subsequently identifying this error.

[2]BookCatalogue. https://play.google.com/store/apps/details?id=com.eleybourn.bookcatalogue

With the guidance of RLDroid, there has been a notable enhancement in both the coverage and the detection of crashes by Monkey. The UTG constructed by RLDroid is useful for automated GUI testing.

C. Threats to Validity

The main threats to external validity lie in the selection of the apps. RLDroid is evaluated on 29 Android apps. Our results may not be applicable beyond the 29 apps to which we have applied RLDroid. To mitigate this threat, we chose apps from two standard testing benchmarks, ANDROTEST [3] and THEMIS [49], which has been used in previous studies [20], [27], [41], [42].

Threats to internal validity are factored into our experimental methodology and they may affect our results. We manually explore apps to check false positive transitions, which is potentially error-prone. To minimize this threat, at least two researchers performed manual checks and compared the experimental results to check for discrepancies.

VII. Related Work

A. Static UTG Construction

There are several studies that utilize static analysis to construct UTG. Among them, A^3E [9] is the first to build a static model of an Android app which constructs Activity Transition Graph (ATG) by data flow analysis [40]. Gator [13] leverage static reference analysis for GUI objects [50] and context-sensitive static analysis of callback methods [51] to construct a Window Transition Graph (WTG) which adds components like menus and dialogs on the ATG. GoalExplorer [15] further extends the WTG by adding other components such as fragments, drawers, and broadcast receivers. There are also some works that employ inter-component communication (ICC) analysis to construct UTG. EPICC [10] is the first to extract ICC information. Based on this, IC3 [11] improves the modeling of ICC objects and enhances extraction capabilities. StoryDroid [7] extends IC3 and adds the fragment and inner class information on ATG. Fax [52] enhances the integrity of ATG through multi-entry ICC analysis. However, due to the weaknesses of reference analysis, using it to obtain UTGs can result in a large number of false positive transitions. On the other hand, static analysis struggles to identify dynamically rendered UI pages. RLDroid leverages static analysis, combined with Android documentation and the app's call graph, to obtain a relatively accurate seed UTG to guide automated dynamic exploration.

B. Dynamic UTG Construction

To improve the accuracy of UTG, some researchers have adopted dynamic exploration for constructing UTG, which includes random strategies [37], [53], [54], model-based methods [14], [20], [55], [56], systematic search [41], [57] and supervised learning based exploration [58]. However, dynamic analysis cannot capture a complete UTG due to its low UI page coverage. Therefore, some studies, such as Fax [52], GESDA [59], StoryDistiller [8], SceneDroid [18] and ICCDROID [42], integrate static and dynamic analysis to construct the UTG. There are also studies that combine dynamic analysis with machine learning to predict transitions between UI pages. Promal [19] uses machine learning to predict feasible transitions in the WTG. ArchiDroid [21] leverages graph neural networks to predict transitions in the ATG based on the graph structural similarity of ATGs within the same category of apps. Different from those works, RLDroid constructs a seed UTG to guide the depth-first dynamic exploration and employs reinforcement learning to enhance the completeness of UTG.

C. Reinforce Learning Based Testing

Recently, some studies [25], [27], [29], [31], [32] have applied reinforcement learning to GUI testing. QBE [25] categorizes states based on the number of widgets on the UI page and learns the value of different types of events from a set of Android apps. Q-testing [27] designs a curiosity-driven exploration strategy to explore previously unvisited UI pages. Based on this, DQT [31] optimizes the problem of comparing similar states during exploration using Deep Q-Network (DQN) and graph neural networks. ARES [29] utilizes Deep Neural Networks (DNN) to learn optimal exploration strategies from previous attempts and implements various reinforcement learning algorithms such as DDPG, SAC, and TD3. Hawkeye [32] uses deep reinforcement learning to learn from historical exploration data and determine the priority of GUI operations related to code changes. However, the goal of these studies is to detect more crashes and the UTGs generated from dynamically testing are usually incomplete. By contrast, RLDroid utilizes a partially constructed UTG for Q-learning initialization and designs an exploration reward to discover more unvisited UI pages.

VIII. Conclusion

In this paper, we propose a novel approach RLDroid to effectively construct UTGs for Android apps. RLDroid constructs a relatively accurate seed UTG through static analysis and leverages it, combined with a depth-first strategy, to guide dynamic exploration. Building on this, we further design a Q-learning-based strategy to enhance dynamic exploration and generate the final UTG. Our extensive experiments on 29 apps demonstrate the effectiveness of RLDroid in constructing UTGs for Android apps. RLDroid identified a total of 871 nodes and 2726 edges without any false positive edges, which significantly outperforms the state-of-the-art GUI modeling techniques. Additionally, the UTGs constructed by RLDroid helped the automated GUI testing tool Monkey improve its average coverage by 60% and discover 52 additional bugs.

Acknowledgment

We thank the anonymous ICPC reviewers for their valuable feedback. This work was partially supported by the National Natural Science Foundation of China (Grant No. 62302327) and the Natural Science Foundation of Jiangsu Province (Grant No. BK20230478).

REFERENCES

[1] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[2] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[3] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440.

[4] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 987–992.

[5] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral analysis of android applications using automated instrumentation," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, 2013, pp. 182–187.

[6] S. N. Dutia, T. H. Oh, and Y. H. Oh, "Developing automated input generator for android mobile device to evaluate malware behavior," in *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*, 2015, pp. 43–43.

[7] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: automated generation of storyboard for android apps," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 596–607.

[8] S. Chen, L. Fan, C. Chen, and Y. Liu, "Automatically distilling storyboard with rich features for android apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 667–683, 2022.

[9] 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.

[10] D. Octeau, P. D. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 543–558.

[11] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. D. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 77–88.

[12] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 280–291.

[13] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 658–668.

[14] T. Gu, C. Sun, X. Ma, C. Cao, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[15] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 2019, pp. 115–127.

[16] J. Yan, S. Zhang, Y. Liu, J. Yan, and J. Zhang, "Iccbot: fragment-aware and context-sensitive icc resolution for android applications," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 105–109.

[17] N. Abolhassani and W. G. Halfond, "A component-sensitive static analysis based approach for modeling intents in android apps," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 97–109.

[18] X. Zhang, L. Fan, S. Chen, Y. Su, and B. Li, "Scene-driven exploration and gui modeling for android apps," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1251–1262.

[19] C. Liu, H. Wang, T. Liu, D. Gu, Y. Ma, H. Wang, and X. Xiao, "Promal: precise window transition graphs for android via synergy of program analysis and machine learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1755–1767.

[20] 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 (ESEC/FSE)*, 2017, pp. 245–256.

[21] Z. Liu, C. Chen, J. Wang, Y. Su, Y. Huang, J. Hu, and Q. Wang, "Ex pede herculem: Augmenting activity transition graph for apps via graph convolution network," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1983–1995.

[22] M. M. Hing, A. v. Harten, and P. Schuur, "Reinforcement learning versus heuristics for order acceptance on a single resource," *Journal of Heuristics*, vol. 13, pp. 167–187, 2007.

[23] A. Usman, M. M. Boukar, M. A. Suleiman, I. A. Salihu, and N.-O. Eke, "Reinforcement learning for testing android applications: A review," in *2023 2nd International Conference on Multidisciplinary Engineering and Applied Science (ICMEAS)*, 2023, pp. 1–6.

[24] Y. Zhao, B. Harrison, and T. Yu, "Dinodroid: Testing android apps using deep q-networks," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–24, 2024.

[25] 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)*, 2018, pp. 105–115.

[26] T. A. T. Vuong and S. Takada, "Semantic analysis for deep q-network in android gui testing." in *SEKE*, 2019, pp. 123–170.

[27] 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.

[28] H. Guo, X. Liu, B. Li, L. Cai, Y. Hu, and J. Cao, "Sqdroid: A semantic-driven testing for android apps via q-learning," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 301–310.

[29] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–29, 2022.

[30] Y. Gao, C. Tao, H. Guo, and J. Gao, "A deep reinforcement learning-based approach for android gui testing," in *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*, 2022, pp. 262–276.

[31] Y. Lan, Y. Lu, Z. Li, M. Pan, W. Yang, T. Zhang, and X. Li, "Deeply reinforcing android gui testing with deep reinforcement learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[32] C. Peng, Z. Lv, J. Fu, J. Liang, Z. Zhang, A. Rajan, and P. Yang, "Hawkeye: Change-targeted testing for android apps based on deep reinforcement learning," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024, pp. 298–308.

[33] (2024) Activity api methods. [Online]. Available: https://developer.android.com/reference/android/app/Activity

[34] (2024) Fragment api methods. [Online]. Available: https://developer.android.com/reference/android/app/Fragment

[35] (2024) Menu api methods. [Online]. Available: https://developer.android.com/develop/ui/views/components/menus

[36] (2024) Dialog api methods. [Online]. Available: https://developer.android.com/reference/android/app/Dialog

[37] (2024) Monkey: A random testing framework. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[38] Z. Chen, J. Liu, Y. Hu, L. Wu, Y. Zhou, Y. He, X. Liao, K. Wang, J. Li, and Z. Qin, "Deuedroid: Detecting underground economy apps based on utg similarity," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 223–235.

[39] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[40] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context,

flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.
[41] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 481–492.
[42] H. Guo, T. Su, X. Liu, S. Gu, and J. Sun, "Effectively finding icc-related bugs in android apps via reinforcement learning," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 403–414.
[43] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.
[44] (2024) Opencv: An image processing framework. [Online]. Available: https://opencv.org/
[45] (2024) Soot: A java optimization framework. [Online]. Available: https://github.com/soot-oss/soot
[46] (2024) Uiautomator. [Online]. Available: https://developer.android.com/jetpack/androidx/releases/test-uiautomator
[47] (2024) Android debug bridge. [Online]. Available: https://developer.android.com/studio/command-line/adb
[48] (2024) Locat command-line tool. [Online]. Available: https://developer.android.com/tools/logcat
[49] 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.
[50] A. Rountev and D. Yan, "Static reference analysis for gui objects in android software," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 143–153.
[51] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 89–99.
[52] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of android applications by constructing activity launching contexts. in 2020 ieee/acm 42nd international conference on software engineering (icse)," *IEEE, 457ś468*, 2020.
[53] 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.
[54] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 204–217.
[55] 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)*, 2017, pp. 23–26.
[56] Y. Ma, Y. Huang, Z. Hu, X. Xiao, and X. Liu, "Paladin: Automated generation of reproducible test cases for android apps," in *Proceedings of the 20th international workshop on mobile computing systems and applications*, 2019, pp. 99–104.
[57] 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.
[58] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1070–1073.
[59] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, "Improving automated gui exploration of android apps via static dependency analysis," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 557–568.